# UNIPLUS+ SYSTEM V
# User Guide

UniSoft
SYSTEMS

PN: 1178-01

# PREFACE

The *UniPlus+* System V User Guide* is a description of the features and a general overview of the capabilities of UniPlus+. Instructions on how to use the system are also included. Not all of the capabilities of the operating system are described or illustrated herein, but enough are described so that a new user can become familiar with its use.

This guide contains seven chapters:

- INTRODUCTION
- BASICS FOR BEGINNERS
- TEXT EDITOR — ED
- VISUAL TEXT EDITOR — VI
- THE SHELL — SH
- THE C SHELL — CSH
- GLOSSARY

Chapter 1, INTRODUCTION, gives beginners an overview of UniPlus+. A beginner should read this chapter before attempting to use the information covered in the other chapters of this guide. An experienced user probably does not need to read this introductory chapter.

Chapter 2, BASICS FOR BEGINNERS, discusses aspects of UniPlus+ that helps the beginning user get started on the system. It also contains a brief explanation of document preparation and programming.

Chapter 3, TEXT EDITOR — ED, is designed to help users get started with text editing and discusses the user's day-to-day needs regarding the use of the text editor **ed**.

---

* UniPlus+ is a trademark of UniSoft Corporation.

Chapter 4, VISUAL TEXT EDITOR – VI, provides the information necessary to use the display-oriented text editor. It is suggested to run the **vi** program while reading this chapter.

Chapter 5, THE SHELL – SH, covers simple commands and procedures and contains helpful information for writing shell scripts.

Chapter 6, THE C SHELL – CSH, outlines the C shell's unique features, commands, procedures, and also contains helpful information for writing shell scripts.

Chapter 7, GLOSSARY, is an alphabetical list of common UNIX† terms. This is a helpful tool for any user.

Throughout this guide, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *UniPlus+ System V Administrator's Manual.* Other references to entries of the form **name**(*N*), where *N* is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section *N* of the *UniPlus+ System V User's Manual.*

† UNIX is a trademark of AT&T Bell Laboratories, Inc.

# CONTENTS

# Chapter 1: INTRODUCTION

CONTENTS

# Chapter 1
# INTRODUCTION

## 1. Overview

It is not intended for this chapter to be a detailed description, rather to give a beginner an overview of UniPlus$^+$, the high performance UNIX operating system, and general instructions on how to begin working on the system. Many of the subjects described are discussed in detail in other sections of this guide or in the *UniPlus$^+$ System V User's Manual.*

In this guide, software programs that can be executed by users are referred to as *programs.* A program that is in some state of execution is referred to as a *process.* The request typed by the user is referred to as a *command* or *command line.* The following graphic conventions are used in the examples:

| | |
|---|---|
| RETURN | Indicates that the user should press the RETURN (carriage return) key on the terminal keyboard. |
| DEL | Indicates that the user should press the key marked DEL, DELETE, or RUBOUT (whichever is appropriate for the terminal being used). |

Throughout this guide, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *UniPlus$^+$ System V Administrator's Manual.* Other references to entries of the form **name**(*N*), where *N* is a number (1 through 6) possibly followed by a letter, refer to entry **name** in section *N* of the *UniPlus$^+$ System V User's Manual.*

## 2. Terminal

In order to log in, the power to the terminal must be turned on and the appropriate switches set. Before you can begin to use the system, you must be connected or dialed up to the system from a full-duplex ASCII terminal. Most terminals have a a half-/full-duplex switch that should be set to full, meaning that input and output can be transmitted simultaneously.

The terminal will also have a speed switch. Common terminal speeds are 10, 15, 30, and 120 characters per second (110, 150, 300, and 1,200 baud); speeds of 240, 480, and 960 characters per second (2,400, 4,800, and 9,600 baud) are also available. Hard-wired terminals are usually set to the correct speed.

If you have a hard-wired terminal, you need only to turn it on and hit the RESET/BREAK or ATTENTION key until the "**login:**" message appears. However, you may need to get an appropriate telephone number from your system administrator to dial up the system. Depending on the type of terminal and communication link, the user may need to press the RESET/BREAK key a couple of times. This is to synchronize your terminal with the system.

### 2.1 Strange Terminal Behavior

Sometimes your terminal acts strangely. For example, each letter may be typed twice (terminal may be in the half-duplex mode) or the RETURN may not cause a line feed or a return to the left margin. The user can often remedy this by logging out and logging back in. If logging back in fails to correct the problem, check the following areas:

| | |
|---|---|
| keyboard | Keys such as caps lock, local, block, etc. should not be in depressed position. |
| dataphone | For terminals connected via phone lines, the baud rate could be incorrect. |
| switches | The rear panel of your terminal normally has several switches used to control terminal operations. These switches should be set to be compatible with the operating system. |

If all else fails, the description of the stty(1) command can be read to determine the appropriate action to take.

## 3. Logging In

UniPlus$^+$ is accessed by the use of a **login**. A **login** name is used by the system to uniquely identify users. Before the user can access the system, the user must be assigned a **login** account by the system administrator.





A **login** name is a unique string of letters (should be all lowercase) and/or numbers that identifies an individual to the system. The **login** name must begin with a letter. If you type uppercase letters, the system will assume that your terminal cannot generate lowercase letters and that you intend all subsequent uppercase input to be treated as lowercase. In many cases, a person's **login** name is their real first name, last name, initials, or nickname. Any string of letters and/or digits can be used as your **login** name, as long as it is *unique* (i.e., different from all other **login** names). Only the first eight characters of a **login** name are used by the system. **Login** names are assigned by the system administrator.

The password is a string of up to 13 characters chosen from a 64-character alphabet (., \, 0-9, A-Z, a-z) that serves to control access to a **login**. The password for a **login** is the main security feature of UniPlus$^+$. Usually, every **login** is assigned a password. When a user logs in to the system, the password (if any) assigned to the **login** being used is requested. Access to the system is not permitted until the correct password is entered. The user can change their password as needed to ensure that others are not accessing their **login** (and consequently their data). Any string of letters, numbers, etc., can be used as a password as long as it is from six to thirteen characters in length and composed of uppercase letters, lowercase letters, numbers, or punctuation.

It is recommended that obvious strings such as the user's social security number, birth date, or other data that could be well known about the user not be used as passwords. If the password is something that is well known about the user, someone could gain access to the user's **login** with little effort. The more unusual your password, the more effective your security.

When communication is established, the system will prompt with:

```
login:
```

The user should type in his/her **login** name followed by a RETURN. After the system digests your **login** name, it will prompt for your password with:

Password:

The user should then type his/her password followed by a RETURN. The system does not echo (print) your password on the terminal as you type it in. This is an extra security measure. If you entered your **login** name and password correctly, the system may print one or more "messages of the day". Following the messages, the system will prompt you with the primary prompt string, which is usually the "$" or "%" symbol. If a mistake is made while logging in or the system administrator has not set up the user's **login** on the system, the following error message is printed:

login incorrect

This error message is followed by the "**login:**" message. The user should attempt to log in again.

## 4. Current Directory

The UniPlus+ file system is arranged in a hierarchy of directories. When the system administrator gave you a user name, s/he also created a directory for you (ordinarily with the same name as your user name, and known as your **login** or *home* directory). When you log in, that directory becomes your *current* or *working* directory, and any file name you create is by default assumed to be in that directory. (When the system assumes something, this is called a *default* value.)

The user may, however, create one or more directories under the home directory. Under a directory or a subdirectory, the user may create files as necessary. The user is the owner of the home directory and all subdirectories created under the home directory. As the owner, the user has full permission to create, alter, and remove (destroy) all files and subdirectories of the home directory. Permissions to have your will with others directories and files will have been granted or denied to you by their respective owners, or by the system administrator.

The user may change from one directory to another by using the **cd** (change directory) command. See **cd**(1) for details.





## 5. Path Names

To refer to files not in the current directory, you must use a path name. Full path names begin with /, which is the name of the *root* directory of the whole file system. After the slash comes the name of each directory containing the next sub-directory, followed by a /, until the file name is reached For example, "/usr/scr/filex" refers to file "filex" in directory "scr", which is itself a subdirectory of "usr". The "usr" directory springs directly from the root directory. See **intro**(2) for a formal definition of *path name.*

If your current directory contains subdirectories, the path names of files therein begin with the name of the corresponding subdirectory (without a prefixed /). With few exceptions, a path name may be used anywhere a file name is required.

Important commands that modify the contents of files are: **cp**, which copies a file (resulting in two identical files); **mv**, which moves the contents of one file into another, removing the first file; and **rm**, which deletes a file or files. To find out the status of files or contents of directories, use **ls**. Use **mkdir** to make directories and **rmdir** to remove directories. These commands are explained in greater detail in Section 1 of the *UniPlus+ System V User's Manual.*

## 6. Entering Commands

The operating system *shell* (command interpreter) serves as the interface between the user and the system. The shell accepts requests from the user in the form of a *command line* and invokes the appropriate program to fulfill the request. The shell prompts (i.e., notifies) the user when it is ready to accept another request.

### 6.1 Command Line Syntax

Commands or requests to the shell are usually in the form of a single line—that is, a string of one or more words followed by a return. This single line request entered following the prompt is referred to as a *command line.* The command line is divided into two major parts—the program name and arguments.

The first word of the command line is the name of the program to be executed. This is referred to as the *command.* All subsequent words

are *arguments* to the command. Arguments are used to provide information required by the program.

Spaces and tabs serve as the delimiters for words on the command line. That is, all characters on the command line up to the first space or tab are interpreted as the command. All characters between the first space (or tab) and the second space (or tab) are the first argument, etc. Thus, the syntax for the command line is:

*command argument argument argument* ... RETURN

### 6.2 Read-Ahead

UniPlus+ has full read-ahead, which means that you can type at any time, even while a program is typing at you. So the user can type several commands one after another without waiting for the first to finish or even begin. Of course, if you type during output, the output will be interspersed with the input characters. However, whatever you type will be saved and interpreted in the correct sequence. There is a limit to the amount of read-ahead, but it is generous and not likely to be exceeded unless the system is in trouble. When the read-ahead limit is exceeded, the system throws away all the saved characters.

### 6.3 Correction and Deletion

All users are likely to make mistakes, especially when typing. Two features are provided to correct command lines. These features are *only* effective for the current line (i.e., you have not ended the line with a return yet).

The first correction feature is the erase character (by default, #). When inputting a command line, the erase character erases the character preceding it. Successive uses of # will erase characters back to, but not beyond, the beginning of the line.

The second correction feature is the kill character (by default, @). The kill character deletes the entire current line.

If the actual # or @ character is needed in a command line, the backslash character (\) preceding it will turn off the "erase last character" or "delete entire line" meaning of the symbol. For example, entering the line





```
mail ann\@uam
```

results in "mail ann@uam".

These default erase and kill characters can be changed; see **stty**(1).

## 7. Programs

### 7.1 Running a Program

Once logged in, you are in direct communication with a program called **sh** (the shell). The shell reads the lines you type, splits them into a command name and its arguments, and executes the command.

A command is simply an executable program. Normally, the shell looks first in your current directory for a given program, and if none is there, then in system directories. There is nothing special about system-provided commands except that they are kept in directories where the shell can find them. You can also keep commands in your own directories and arrange for the shell to find them there.

The command name is the first word on an input line to the shell; the command and its arguments are separated from one another by space and/or tab characters. The command sequence is followed by typing a RETURN.

When a program terminates, the shell will ordinarily regain control and display a prompt sign to indicate it is ready for another command. The shell has many other capabilities, which are described in detail in sh(1) and **csh**(1) of the *UniPlus+ System V User's Manual.*

### 7.2 Stopping a Program

Most programs can be stopped by typing the character DEL (perhaps called DELETE or RUBOUT on your terminal). The INTERRUPT or BREAK key found on most terminals can also be used. In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone with the talk button depressed will also stop most programs.

### 7.3 Writing a Program

To enter the text of a source program into the UniPlus+ file system, use the text editors **ed**, **ex** or **vi**. The principal languages available under UniPlus+ are C [see **cc**(1)] and assembly language [see **as**(1)]. After the text has been entered with the editor and written into a file, you can give the name of that file to the language processor as an argument. Normally, the output of the language processor will be left in a file in the current directory named "a.out." If that output is precious, use **mv** to give it a less vulnerable name. If the program is written in assembly language, you will probably need to load library subroutines (see **ld**(1)). C language calls the loader automatically.

When you have finally gone through this entire process without provoking any diagnostics, the resulting program can be run just like you would run any other command.

Your programs can receive arguments from the command line just as system programs do [see **exec**(2)].

## 8. Text Processing

You can enter text with the editor **ed** or **ex**, or with the visual text editor, **vi**, which is screen oriented and more suitable for writing documents.

Commands often used to write text on a terminal are: **cat** , **pr**, and **nroff/troff**. The **cat** command dumps text on the terminal with no processing at all. The **pr** command paginates the text, supplies headings, and has a facility for multi-column output. **Nroff/troff** is an elaborate text formatting program requiring careful forethought in entering both text and formatting commands. **Troff** is very similar to **nroff**, but can accept instructions to produce its output on a phototypesetter (it was used to typeset this manual). There are several "macro" packages (specifically, the **mm** package) that significantly ease the effort required to use **nroff** and **troff**. The *UniPlus+ System V Documentation Processing Guide* provides detailed tutorials in the above text processing programs.

## 9. Mail

After logging in, the user may sometimes get the following message:





You have mail.

JniPlus+ provides a postal system so you can communicate with other isers of the system. To read your mail, type the following command:

mail

'our mail will be printed, one message at a time, most recent message irst. After each message, **mail** waits for you to say what to do with it. 'he two basic responses are **d**, which deletes the message, and tETURN, which prints the next message but does not delete the previ- ous message. Other responses and features are described in **mail**(1) in he *UniPlus+ System V User's Manual.*

To send mail to another user, type the following:

**mail** *user-login-name*
one or more lines of message
**CONTROL-d**

The "CONTROL-d" sequence, often called End-Of-File (EOF), is used o mark the end of input from a terminal.

For practice, try sending mail to yourself. (This is not as strange as it night sound — mail to oneself is a handy reminder mechanism.)

## 10. Writing to Other Users

At some point, out of the blue will come a message like

Message from diane tty07...

which is accompanied by a startling beep on terminals that have the capability to beep. It means that Diane (diane) wants to talk to you, but unless you take explicit action, you will not be able to talk back. To respond, type the following command:

write diane

This establishes a 2-way communication path. Now whatever diane types on her terminal will appear on yours and vice versa. However, if you are in the middle of some program, you must get back to a state

where you are talking to the command interpreter. Normally, whatever program you are running has to terminate or be terminated. If you are editing, you can escape temporarily from the editor by typing a CONTROL-z. This will put the file you are editing in the background so that you can execute the **write** program. To resume your editing job, type **fg** which will bring the job to the foreground.

If you are printing and do not want this message in your printout or you simply do not want to be disturbed, enter the following:

```
mesg n
```

A protocol is needed to keep what you type from getting garbled up with what diane types. Typically, a sequence like the following is used:

| Diane | Ralph |
|---|---|
| types "write ralph" and waits | types "write diane" and waits |
| types a message of as many lines as necessary [when she is ready for a reply, she signals it by typing **(o)** which stands for "over"] | types a reply, also terminated by **(o)** |

This cycle repeats until someone gets tired; s/he then signals her or his intent to quit with **(oo)** for "over and out".

To terminate the conversation, each side must type a CONTROL-d character alone at the beginning of a line (DELETE also works). When the other person types CONTROL-d, you will get the message **EOF** on your terminal.

If you write to someone who is not logged in or who does not want to be disturbed, you will be told. If the target is logged in but does not answer after a decent interval, simply type a CONTROL-d.

## 11. On-line Manual

The *UniPlus+ System V User's Manual* and the *UniPlus+ System V Administrator's Manual* are kept on-line. If you get stuck on something





and can not find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type "**man** *command-name*". Thus to read up on the **who**(1) command, type

```
man who
```

and

```
man man
```

tells all about the **man**(1) command.

## 12. Logging Out

After completing your work, it is best to log off the system. Before logging off, you should have received the prompt symbol from the system. That is, all your commands have been completed, and the system is ready for another command.

A common method for logging off is accomplished by typing an American Standard Code for Information Interchange (ASCII) End Of Text (EOT) character which is sometimes called the End-Of-File (EOF). On most terminals, the EOT character is generated by holding down the CONTROL key and pressing the lowercase "d" key once. This is also referred to as a **CONTROL-d**. Regardless of the terminal type, the power to it should be turned off when the terminal is no longer needed. For a terminal connected via a phone line, you should hang up the phone.

Another way to log off the system is by simply typing:

```
logout
```

# Chapter 2: BASICS FOR BEGINNERS

CONTENTS

# Chapter 2

# BASICS FOR BEGINNERS

## 1. Day-to-Day Use

### 1.1 Creating Files — The Editor

If you have to type a paper, a letter, or a program, how do you get the information into the machine? These tasks can be performed using the UniPlus+ "text editor". See **ed**(1), **vi**(1) and Chapters 3 and 4 of this guide for a detailed description.

The UniPlus+ text editor operates on a file. A file is a collection of information stored in the machine. The following describes how to make some *files.* For example, to create a file called "junk" with text in it, do the following:

```
ed junk                    (invokes the text editor)
a                          (command to "ed" to add text)
now type in
whatever text you want ...
.                          (signals the end of text addition)
```

The "." signals the end of adding text and must be at the beginning of a line by itself. Do not forget it, for until it is typed, no other **ed** commands will be recognized—everything you type will be treated as text to be added. Also note that no system prompt appears while you are appending, inserting, or changing text in the text editor.

After a file exists, the user can edit the text which was typed in—correct spelling mistakes, rearrange paragraphs, etc.

Finally, the user must write the information typed into a file with the editor command:

```
w
```

**Ed** responds with the number of characters it wrote into the file "junk."

Nothing is stored permanently in the "junk" file until you type **w**. If you are editing a file and hang up before typing **w**, the changes are not stored in the working file. Instead, they are saved in a file called "ed.hup" which you can continue working with at the next editing session. But after typing **w**, the information is there permanently. You can retrieve it any time by typing:

```
ed junk
```

Type **q** to quit the editor. (If you try to quit without writing, **ed** prints **?** to remind you to save the file. Typing **q** again exits you from the editor without saving the file, if that's really what you want to do.) Now create a second file called "temp," following the procedures you used to create "junk." You should now have two files, "junk" and "temp."

### 1.2 What Files Are Out There?

The **ls** command lists the names (not contents) of the files the system knows about. If you type

```
ls
```

the response is:

```
junk
temp
```

which are the two files you just created.

The file names are automatically listed in alphabetical order, but you can change this. For example, typing

```
ls −t
```

lists the files in the order in which they were last changed, most recent first. The **−l** option gives a "long" listing and is used as follows

```
ls −l
```

to produce something like

```
-rw-rw-rw- 1 bwk  bsk 41 Jul 22 02:56 junk
-rw-rw-rw- 1 bwk  bsk 78 Jul 22 12:57 temp
```

The date and time is the date and time of the last change to the file. "41" and "78" are the number of characters (which should agree with





the numbers you got from **ed**). "bwk" is the owner of the file, i.e., the person who created it. "bsk" identifies the group associated with "bwk". "-rw-rw-rw-" determines who has permission to read, write, or execute the file. In this case the owner, group, and others all have permission to read (r) and write (w). There is no permission for anyone to execute (x). The first character in "-rw-rw-rw-" is a "-" which indicates this is a data file. A "d" as the first character indicates a directory. The remaining nine characters are divided into three sets of permissions. Each set consists of three characters. The three sets correspond to the permissions of the owner, group, and all other users.

Options can be combined: **ls −lt** has the same listing as **ls −l** but is sorted into time order. You can also specify the files you're interested in, and **ls** will list only the information about them. More details can be found in ls(1).

Optional arguments that begin with a minus sign (like **−t** and **−lt**) are a common convention for UniPlus+ programs. In general, if a program accepts such optional arguments, they precede any file name. It is also vital that you separate the various arguments with spaces: **ls−l** is not the same as **ls −l** since the command **ls** must be separated from its argument **−l** by a space.

### 1.3 Printing Files

Now that you've created a text file, how can you print it? There are several ways to print a file. If you want to print your file on the screen, one simple way is to use the editor. The editor prints as follows:

```
ed junk
1,$p
```

**Ed** will reply with the count of the characters in "junk" and then print all the lines in the file. The user can also select the parts of a file to print as follows:

```
ed junk
20,35p
```

which will print only lines 20 through 35.

There are times when it's not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time; and sometimes you want to print several, one after the other. Finally, **ed** is not designed to print files on a printer. Here are a couple of alternatives.

The simplest of all the printing programs is **cat**. Cat simply prints the contents of all the files named in the order listed. Thus the files are concatenated (joined together) and printed. For example:

```
cat junk
```

prints one file, and

```
cat junk temp
```

prints two files. The files are simply concatenated onto the terminal.

The **more** command stops after each page is printed on your screen. This keeps **cat** from scrolling the text off the screen before you can read it. For example, to use **more**, type the following

```
cat junk temp | more
```

(| is called a pipe and is described later in this chapter.)

The **pr** command produces formatted printouts of files. As with **cat**, **pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number, and file name at the top of each page. It also will give extra lines to skip over the fold in the paper when you print a file on the printer.

Thus,

```
pr junk temp | lpr
```

prints "junk" neatly, then skips to the top of a new page and prints "temp" neatly.

**Pr** can also produce multicolumn output. Typing

```
pr -3 junk
```

prints "junk" in 3-column format. You can use any reasonable





number in place of "3," and **pr** will do its best. The **pr** command has other capabilities also. See **pr**(1) for more information.

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are other programs that send your files to a hard copy printer. See **lp**(1) and **lpr**(1) for more information.

### 1.4 Moving Files Around

You are ready for bigger things after gaining experience in creating and printing files. For example, you can move a file from one place to another (which amounts to giving it a new file name), like this:

```
mv junk precious
```

This means that what used to be named "junk" is now named "precious." Typing Is results in the following:

```
precious
temp
```

The contents of "junk" are now in "precious." Notice that the "junk" file no longer exists. If you move a file to one that already exists, the already existing file contents are lost *forever*.

If you want to make a copy of a file (i.e., to have two versions), use the **cp** command as follows:

```
cp precious temp1
```

This makes a duplicate copy of "precious" in "temp1."

When you are finished creating and moving files, the files can be removed from the file system with the **rm** command. The command is used as follows:

```
rm temp temp1
```

This will remove both the "temp" and "temp1" files.

You will get a warning message if one of the named files is not there, but otherwise **rm**, like most UniPlus+ commands, does its work silently. There is no prompting or response, and error messages are short. This terseness is sometimes disconcerting to newcomers, but experienced users prefer it.

### 1.5 What's in a File Name

So far we have used file names without ever saying what is a legal name, so it is time for a couple of rules. First, file names are limited to 14 characters. Second, although any character can be used in a file name, common sense dictates sticking to ones that are visible and avoiding characters that could have other meanings. We have already seen, for example, that in the **ls** command, **ls −t** lists in time order. So, if a file were named "−t" you would have a tough time listing it by name. In addition to the minus sign, there are other characters which have special meaning. To avoid pitfalls, use only letters, numbers, and the period until you are familiar with the system.

Suppose you are typing a large document, like a book. Logically, this can be divided into many small pieces, like chapters and perhaps sections. Physically, it must be divided too, for **ed** will not handle really big (over 90,000 characters) files. Thus the document should be typed as many files. One possible method is to have a separate file for each chapter as follows:

```
chap1
chap2
etc. ...
```

Another method is breaking each chapter into several files as follows:





```
chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...
```

It can now be determined at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice user. To print the whole book, you could type the following:

```
pr chap1.1 chap1.2 chap1.3 ...
```

Using the **pr** command like this would be tiring and possibly lead to mistakes. Fortunately, there is a shortcut. You can type:

```
pr chap*
```

The * means "anything at all," so this translates into "print all files whose names begin with *chap* listed in numerical and then alphabetical order."

This shorthand notation is not a property of the **pr** command by the way. It is system-wide, a service of the program that interprets commands—the "*shell*," **sh** and **csh**. The files in the book can be listed by using

```
ls chap*
```

which produces the following:

```
chap1.1
chap1.2
chap1.3
...
```

The * can be in any position in a file name and can occur several times. Thus, typing

```
rm *junk* *temp*
```

removes all files that contain "junk" or "temp" as any part of their name. * by itself matches every file name, so

```
pr *
```

prints all your files (in alphabetical order), and

```
rm *
```

removes *all files.* (Before using **rm ***, be sure you want to remove all your files!)

The * is not the only pattern-matching feature available. To print only chapters 1 through 4 and 9, type the following:

```
pr chap[12349]*
```

The [...] means to match any of the characters inside the brackets. A range of consecutive digits can be abbreviated as follows:

```
pr chap[1-9]*
```

Letters can also be used within brackets. **[a-z]** matches any character in the range *a* through *z.*

A ? matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter ("chap1.1," "chap2.1," etc.).

Of these niceties, * is the most useful. The others are frills, but worth knowing.

If the special meaning of *, ?, etc., needs to be turned off, enclose the entire argument in single quotes as follows:





```
ls '?'
```

Some examples of this are shown in the following paragraphs.

### 1.6 Directories and Pathnames

When you first create the file called "junk," how does the system know that there is not another "junk" somewhere else, especially since the person in the next office could also be reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to that particular user. When you **login**, you are *in* your directory. Unless you take special action when creating a new file, the new file is made in the directory that you are currently in. This is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a tree with your files located several branches into the tree. It is possible for you to "walk" around this tree and find any file in the system by starting at the root of the tree and walking along the proper set of branches. You can also start at your present location and walk toward the root.

Try the latter first. The basic tool is the command **pwd** (print working directory) which prints the name of the directory you are currently in.

Although the details vary, **pwd** prints something like:

```
/usr/your-name
```

This indicates that you are currently in the directory "your-name," which is in turn in the directory "usr," which is in turn in the root directory called by convention just "/". (Even if it is not called "/usr" on your system, the message will be something analogous. Recognize any differences between your machine's pathname and the standard setup and make the corresponding changes to the following command lines.)

If you now type

```
ls /usr/your-name
```

the results should be exactly the same list of file names as a plain **ls**. With no arguments, **ls** lists the contents of the current directory. Given the name of a directory, it lists the contents of that directory.

Next, try the following command:

```
ls /usr
```

This should print a long series of names, among which is your own **login** name "your-name." On many systems, "usr" is a directory that contains the directories of all the normal users of the system.

The next step is to try the following:

```
ls /
```

The response should be something like this:

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

If "junk" is still in your directory, enter the following:

```
cat /usr/your-name/junk
```

The name

```
/usr/your-name/junk
```

is called the *pathname* of the file that is normally thought of as "junk." The pathname represents the full name of the path as followed from the root through the tree of directories to get to a particular file. It is a rule in the UniPlus+ system that the pathname can be substituted anywhere an ordinary file name can be used.





This is not too exciting if all the files of interest are in your own directory; but if you work with someone else or on several projects concurrently, it becomes handy. For example, your friends can print your book by typing the following:

```
pr /usr/your-name/chap*
```

Similarly, you can find out what files your neighbor has by entering:

```
ls /usr/neighbor
```

The "neighbor" represents the **login** name of your neighbor. You can copy one of your neighbor's files as follows:

```
cp /usr/neighbor/their-file your-file
```

If a file owner does not want someone else to have access to the owner's files, privacy can be arranged. Each file and directory has read-write-execute (rwx) permissions for the owner, a group, and everyone else, which can be set to control access. See **ls**(1) and **chmod**(1) for details. Most users find openness of more benefit than privacy (most of the time).

As a final experiment with pathnames, try the following:

```
ls /bin /usr/bin
```

Do some of the names look familiar? When a program is run by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically does not find it), then in "/bin" and finally in "/usr/bin." There is nothing magic about commands like **cat** or **ls**, except that they have been collected into places where they are easy to find and administer.

It is possible for two or more users to work regularly with common information in the same document. If this common document is located in the one directory, the users can change the current working directory as follows:

```
cd full-path-name
```

Now you are ready to edit files in this directory.

Another method of working on the same document is to locate the files in your friend's directory and **login** as your friend. This defeats the accounting purpose of individual **logins**. If you are already logged in as yourself and want to work in a friend's files, change the current working directory as follows:

```
cd /usr/your-friend
```

Now when a file name is used in something like **cat** or **pr**, the command refers to the file in your friend's directory. Changing directories does not affect any permissions associated with a file. If you cannot access a file, get the owner to change permissions via **chmod**. Of course, if you forget what directory you are in, type

```
pwd
```

to find out.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when writing your book, you might want to keep all the text in a directory called "book." A directory can be made using the **mkdir** command. The "book" directory is made as follows:

```
mkdir book
```

You can access the "book" directory as follows:

```
cd book
```

If you logged in as yourself, the pathname of "book" is:

```
/usr/your-name/book
```

To remove the "book" directory, type:





```
rm book/*
rmdir book
```

or

```
rm -r book
```

The "**rm book/***" command removes all files in the "book" directory, and then the "**rmdir book**" command removes the empty directory. The "book" directory must be empty before the **rmdir** command will work. The "**rm -r book**" command deletes the entire contents of the "book" directory and then removes the "book" directory itself. **WARNING:** Be extra careful when using the "**rm -r**" command.

You can go up one level in the tree of files by typing:

```
cd ..
```

The ".." pattern is the name of the parent of whatever directory you are currently in. "." is an alternate name for the directory you are in.

### 1.7 Using Files for Input and Output

Most of the commands used so far produce output on the terminal. Other commands, like the editor, take input from the terminal. The terminal can be replaced by a file for input and output.

As an example,

```
ls
```

lists files on your terminal. But if you enter

```
ls > filelist
```

a list of your files is placed in the file "filelist" (which is created if it does not already exist or overwritten if it does). The symbol > means "put the output of this command in the following file rather than displaying it on the terminal." Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of **cat** in a file:

```
cat f1 f2 f3 > temp
```

Another symbol that operates very much like > does is >>. >> means "add to the end of." That is,

```
cat f1 f2 f3 >> temp
```

means to concatenate "f1," "f2," and "f3" to the end of whatever is already in "temp" instead of overwriting the existing contents. If "temp" does not exist, you will receive an error message.

In a similar way, the symbol < means to take the input for a program from the following file instead of from the terminal. Thus, you could make up a script of commonly used editing commands and put them into a file called "script." The script could then be run on a file by typing:

```
ed file < script
```

Another example is preparing a letter in file "let." The letter could then be sent to several people as follows:

```
mail adam eve mary joe < let
```

### 1.8 Pipes

One of the novel contributions of the UniPlus+ system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes—a pipeline.

For example,

```
pr f g h
```

will print the files "f," "g," and "h," beginning each on a new page. Instead of printing the files separately, the files can be printed together as follows:

```
cat f g h > temp
pr < temp
rm temp
```

This method is more work than necessary. To take the output of **cat** and connect it to the input of **pr**, use the following pipe:





```
cat f g h | pr
```

The vertical bar | means to take the output from **cat**, which would normally have gone to the terminal, and put it into **pr**.

There are many other examples of pipes. For example,

```
ls | pr -3
```

prints a list of your files in three columns. The program **wc** counts the number of lines, words, and characters in its input; while the **who** command prints a list of users currently logged on the system.

Thus typing

```
who | wc -l
```

tells how many people are logged on.

```
ls | wc -l
```

counts your files.

Most programs that read from the terminal can read from a pipe instead. Most programs that write on the terminal can write on a pipe instead. There can be as many commands in a pipeline as desired.

### 1.9 The Shell

The mysterious *shell* mentioned previously is actually the **sh** or **csh** command interpreter. The shell is the program that translates what you type into commands and arguments. The shell also looks after translating *, etc., into lists of file names, and <, >, and | into changes of input and output streams.

The shell has other capabilities as well. For example, you can run two programs with one command line by separating the commands with a semicolon. The shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; who
```

performs both commands before returning with a prompt character.

More than one program can run *simultaneously* if desired. This is beneficial when doing something time-consuming, like using the editor script. Running programs in the background prevents waiting around for the results before starting something else. When you type:

ed file < script &

the ampersand (&) at the end of the line means "start running this command, then return a prompt immediately." That is, don't wait for the command to complete. Thus the script begins running in the background, but you can do something else at the same time.

When a command is initiated with &, the system replies with a number called the process number. Programs running in the background can be terminated as follows:

kill *process-number*

The process number identifies the command to stop. If you forget the process number, the **ps** command lists the process number for all programs you are running. (Typing **kill 0** kills all your processes.) If you are curious about other people, **ps −a** provides information about all *active* programs that other users are running.

To start three commands that execute in the order specified in the background, enter the following:

(command-1; command-2; command-3) &

A background pipeline can be started as follows:

command-1 | command-2 &

Just as the editor can get its input from a file instead of from the terminal, the shell can read a file to get commands. For example, suppose you want to perform a sequence of actions after every **login** such as:

- Set the tabs on the terminal
- Find out the date
- Find out who's on the system.





The three commands (**tabs**, **date**, and **who**) could be put in a file called "startup." The "startup" file would then be run as follows:

sh startup

This instruction commands the machine to run the shell with the file "startup" as input. The effect is the same as typing the contents of "startup" on the terminal.

If this is a regular thing, you can get around typing **sh** every time by typing the following command only once:

chmod +x startup

To run the sequence of commands thereafter, you only needs to type:

startup

The **chmod** command marks the file as being executable. The shell recognizes this and runs it as a sequence of commands.

If you want "startup" to run automatically every time you **login**, create a file in your **login** directory called ".profile" and place in it the line "startup." Upon logging in, the shell gains control and executes the commands found in the ".profile" file. We will get back to the shell in the section on programming.

## 2. Document Preparation

UniPlus+ is used extensively for document preparation. There are two major formatting programs (programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, etc.). The **nroff** (pronounced "en-roff") program produces output on terminals and line-printers. The **troff** (pronounced "tee-roff") program produces output on a phototypesetter, which produces very high-quality output on photographic paper. This document was formatted with **troff**.

### 2.1 Formatting Packages

The basic idea of **nroff** and **troff** is that the text to be formatted contains within it *formatting commands* that indicate in detail how the

formatted text is to look. For example, there may be commands that specify how long lines are, whether to use single or double spacing, and the running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several *packages* of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multicolumn output, etc., with little effort and without having to learn **nroff** and **troff.** These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

This section provides a brief description of the "memorandum macros" package known as **mm.** Formatting requests typically consist of a period and one or two uppercase letters, such as

.TL

which is used to introduce a title or

.P

to begin a new paragraph.

The text of a typical document looks something like this:

```
.TL
title
.AU "author information"
.MT "memorandum type"
.P
Enter text ---
---
.P
More text ---
---
.SG "signature"
```

Lines that begin with a period are the formatting macro requests. For example, **.P** starts a new paragraph. The precise meaning of **.P** depends on the output device being used (typesetter or terminal, for instance). For example, **−mm** normally assumes that a paragraph is preceded by a space—one line in **nroff,** and one-half line in **troff,** and the first word is indented. You can change these rules, but they are





changed by changing the interpretation of **.P,** not by retyping the document.

To actually produce a document in standard format using **−mm,** use the command

**troff −mm** *files* ...

for the typesetter and

**nroff −mm** *files* ...

for a terminal. The **−mm** argument tells **troff** and **nroff** to use the manuscript package of formatting requests. There are several similar packages; check with a local expert to determine which ones are used on your machine. The proper terminal filter for the terminal should be used in the command line. The terminal filter option is indicated by **−T** followed by the terminal type. The terminal types are known by various UniPlus+ system utility calls found in the *UniPlus+ System V User's Manual.*

### 2.2 Supporting Tools

In addition to the basic formatters, there are supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the *UniPlus+ System V User's Manual* and check with other UniPlus+ users for other possibilities.

Both **eqn** and **neqn** (see **eqn**(1) for more information) programs let you integrate mathematics into a document in an easy-to-learn language that resembles the way you would speak it aloud.

For example, the **eqn** input

```
sum from i=0 to n x sub i ~=~ pi over 2
```

produces the output

$$\sum_{i=0}^{n} x_i = \frac{\pi}{2}$$

The program **tbl** provides an analogous service for preparing tables. The **tbl** program does all the computations necessary to align complicated columns with elements of varying widths.

The **spell** program detects possible spelling mistakes in a document. The **spell** program compares the words in your document to a dictionary (stored in memory) and prints those words that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a good job.

The **grep** program looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

```
grep 'ing$' chap*
```

finds all lines that end with the letters "ing" in the files "chap*." The "$" indicates that the pattern to search for is at the end of the line, whereas "^" indicates that the pattern to search for is at the beginning of a line. (It is almost always good practice to put single quotes around the pattern to search for in case it contains characters like * or $ that have a special meaning to the shell.) The **grep** program is often used to locate the misspelled words detected by the **spell** program.

The **diff** program prints a list of the differences between two files, so that two versions of something can automatically be compared. This is a vast improvement over proofreading by hand.

The **wc** program counts the words, lines, and characters in a set of files. The **tr** program translates characters into other characters. For example, this translates uppercase into lowercase:

```
tr [A-Z] [a-z] < input > output
```

The **sort** program sorts files, while **cxref** makes cross-references. The **ptx** program makes a permuted index (keyword-in-context listing). The **sed** program is like **ed** but can be used with long files. These programs are for more advanced users and are not limited to document preparation. Put them on your list of things to learn.

### 2.3 Hints for Preparing Documents

Most documents go through several versions (always more than expected) before they are finally finished. You should do whatever possible to make the job of changing them easy.





First, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting, and rearranging sentences, these precautions simplify any editing needed later.

Keep the individual files of a document down to modest size, perhaps 10,000 to 15,000 characters. Larger files edit more slowly. If you make a dumb mistake, it is better to clobber a small file than a big one. Split the files at natural boundaries in the document for the same reasons that you start each sentence on a new line.

The second aspect of making changes to documents easy is not to commit to the formatting details too early. One of the advantages of formatting packages is permitting format decisions to be delayed until the last possible moment. Until a document is printed it is not even decided whether it will be typeset or printed out on a line printer.

As a rule of thumb, a document should be produced in terms of a set of requests or commands (macros) for all but the most trivial jobs. The macros used should then be defined either by using one of the existing macro packages (the recommended way) or by defining your own **nroff** and/or **troff** macros. As long as the text is entered in some systematic way, it can always be cleaned up and formatted by a judicious combination of editing commands and macro definitions.

## 3. Programming

We will not teach any of the programming languages available, but a few words of advice are in order. One of the reasons why the UniPlus$^+$ system is a productive programming environment is that there is already a rich set of tools available. Facilities like pipes, input/output redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing a program completely from scratch.

### 3.1 Shell Programming

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

```
cat ...    collect the files
|tr ...    put each word on a new line
|tr ...    delete punctuation, etc.
|sort      into dictionary order
|uniq      discard duplicates
|comm      print words in text but not in dictionary
```

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type:

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

The same job can be performed much more easily. One way is to type

```
ls chap* > temp
```

to get the list of file names into a file called "temp." The "temp" file is then edited using global commands as follows:

```
1,$ s/^.*$/e &\
1p\
$p/
```

The results are written into the "script" file (**1,$ w script**) and then the following command is entered:

```
ed < script
```

This will produce the same output as the laborious hand typing. Another method is using shell loops to repeat a set of commands over and over again for a set of arguments as illustrated below:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable **i** to each file name in turn, then does the command. This command can be entered at the terminal or put in a file for later execution. Before the file can be executed, it may be necessary to change the mode by entering the following:

```
chmod +x filename
```

An option often overlooked by new users is that the shell is itself a programming language, with variables, control flow **if-else, while, for, case,** subroutines, and interrupt handling. Since there are many building-block programs, the user can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in Chapters 5 and 6 in this guide.

### 3.2 Programming in C

The C language is a reasonable choice of a programming language when undertaking anything substantial. Everything in the UniPlus+ system is based on the C language. The system itself is written in C, as are most of the programs that run on the system. The C language is also an easy language to use once you get started. The C language is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how to do input/output and similar functions.

Most input and output in C is best handled with the standard input/output library, which provides a set of I/O functions that are in a compatible form with most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library. (Refer to Section 3 of the *UniPlus+ System V User's Manual.*)

There are several supporting programs that go with C. The **lint** program checks C programs for potential portability problems and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file), the **make** program allows you to specify the dependencies among the source files and the processing steps needed to make a new version. The program then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The C compiler provides a limited statistical service, so a user can find where programs spend their time executing and what parts of a program are worth optimizing. Compile the programs with the **−p** option; after the test run, use the **prof** command to print a program execution profile. The command **time** will give the gross run-time statistics of a program, but the times are not very accurate or reproducible.

# Chapter 3: TEXT EDITOR — ED

CONTENTS

# Chapter 3

# TEXT EDITOR — ED

## 1. Introduction

This is a tutorial to help beginners get started with text editing. UniPlus+ has three text editors: **ed**, **ex**, and **vi**. Of these, **ed** is considered the easiest to learn; however, many users prefer **vi**. We recommend reading this chapter first before going on to **vi**. The **ex** program is the root of the text editors and is used mostly by systems programmers and persons very familiar with **ed**.

Although this chapter does not cover everything about the text editor **ed**, it does discuss enough for most day-to-day needs. This includes:

- Printing, appending, changing, deleting, moving, and inserting text.
- Reading from and writing to files.
- Searching for text.
- Making substitutions.
- Making changes throughout a file automatically.
- Using some special characters for easier editing.

## 2. General

The **ed** program is a text editor—an interactive program for creating and modifying text. The text can be a document (like this one), data for a program, instructions for the computer, etc.

Do the exercises. They illustrate techniques discussed in the text. A summary of commands appears at the end of this chapter.

### 2.1 Disclaimer

We cover only a few **ed** commands. (although we include the most common commands). Also, we assume that you know how to log onto UniPlus+ and understand what a file is. For more information about UniPlus+, refer to the Chapters 1 and 2 in this guide.

You also need to know how to end a line on your terminal. On most terminals, you will end a line by pressing RETURN or the newline key.

We refer to this key as RETURN in this chapter.

## 3. Getting Started

To follow along with the tutorial, you should be logged onto UniPlus+ and see the *prompt* character, usually a $ or %.

To start the editor, type:

```
ed
```

and press RETURN.

### 3.1 Creating Text

We describe creating text first. Making changes and corrections is described later.

When you start **ed**, it presents you with an empty file. This is similar to starting to write on a blank piece of paper. To put text in the file, you must type it in or retrieve it from another file. We will begin by the initial typing in of some text and later will describe reading in files.

The text we are working on in **ed** is kept in a buffer. The buffer is like the piece of paper on which we write things, then change some of them, and finally file the whole thing away for another day.

You direct **ed** by typing instructions called *commands*. Most commands consist of one lowercase letter. Each command is typed on a separate line. **Ed** doesn't print any messages in response to most commands.

Once in the editor (by typing **ed** and RETURN), to start adding text, type

```
a
```

on a line by itself and press RETURN. (**a** means "append (or add) text lines to the buffer as typed in.") Type the text you want to add, like this:





```
a
Now is the time
for all good men
to come to the aid of their party.
.
```

As shown in the last line of this example, we stop appending by typing a period on a line by itself and pressing RETURN. This tells **ed** that you have finished adding text and want to give a new command. (Even experienced users sometimes forget to type the period when they have finished adding text. If **ed** seems to be ignoring your commands, type . on a line by itself. You may then find some command lines in your text, which you will have to take out later.)

After you finish appending, the buffer contains these three lines:

```
Now is the time
for all good men
to come to the aid of their party.
```

The **a** and the . are not included since they are commands and not text.

To add more text, type **a** and continue typing.

### 3.2 Error Messages

If **ed** doesn't understand something you type, it prints

```
?
```

on the screen.

This is pretty cryptic, but with practice, you'll learn to recognize the mistake. For some assistance, type

```
h
```

The **h** (help) command explains the most recent ?.

### 3.3 Writing a Text File

After you have added some text to your file, you will want to save it. The **w** (write) command writes the contents of the buffer onto a file. For example, type

**w** *filename*

with *filename* as the name of the file you want your text to be saved to. This copies the buffer's contents into the specified file (destroying any previous information in that file). For example, if we wanted to save the text we created in a file called "junk," we would type:

w junk RETURN

Leave a space between **w** and the file name. In our example, **ed** responds with:

68

**Ed** prints the number of characters it wrote into the file. (Blank spaces and end-of-line characters are included in the character count.) Writing a file just makes a copy of the text—the buffer's contents are not disturbed. This is similar to making a carbon copy of the piece of paper you're writing on and putting this copy in a file folder—it does not change the original, it simply duplicates it. This is an important point. **Ed** only works on a copy of a file (by placing it in the buffer), not the file itself. The file doesn't change until you write your changes to it with the **w** command. (It's a good idea to write your text onto a file at least every hour. If the system crashes or you make some horrible mistake, you will lose all the text in the buffer, but any text in a file will be safe.)

### 3.4 Leaving ed

To leave **ed**, first save your text by writing it onto a file with the **w** (write) command and then type **q** (for quit). For example, in the editing session described above the following is what will appear on the terminal screen:





```
ed                                  (initialize the command)
a                                   (append )
Now is the time                     (text)
for all good men                    (text)
to come to the aid of their party.  (text)
.                                   (end append)
w junk                              (write to file "junk")
68                                  (character count system response)
q                                   (quit)
```

The system will respond with the prompt character.

When you leave **ed** your buffer is destroyed, which is why you write your changes to a file before quitting. Actually, **ed** will print

?

if you try to quit without writing. If you want to save the additions to a file, type **w** and press RETURN. If you don't want to save your additions, typing **q** again gets you out of **ed** without saving any changes since the last **w** command.

EXERCISE 1

Enter **ed** and create some text using the append command, **a**

```
ed
a
This is Exercise 1
to show how to create a file
using the text editor ed.
.
```

No prompt appears in the text editor. When you have finished adding text, type "**w sometext**" and press RETURN to save your text in a file called "sometext." Type **q** to leave **ed**. When you are out of **ed**, you will see the system prompt **$**. To print the file you just created, type

more sometext RETURN

### 3.5 Editing Text Files

After you have created and saved a file, you may want to edit it. The **e** (edit) command retrieves a file you have saved and places it in the buffer. To edit a file, type

**e** *filename* RETURN

If you had saved a file called "junk" as described above and wanted to edit it, you would type

```
ed
e junk RETURN
```

**Ed** retrieves "junk," places it in the buffer, and prints the character count:

```
68
```

When you use **e** to edit a file, **ed** replaces the contents of the buffer with the new file. If you don't save the buffer before using **e**, **ed** destroys the old contents of the buffer.

If you read a file into the buffer with **e**, **ed** uses that file name when you type **w** to write the file. For example, an editing session might look like this:

```
ed
e filename
[editing session]
.
w
q
```

The file you edited is automatically save in *filename* when you type **w**. You can find out which file **w** will write to using the **f** (file) command.

### 3.6 Reading Text

Sometimes you want to add a file to the buffer without destroying anything already in the buffer. This is done with the **r** (read) command:





**r** *filename*

adds the contents of *filename* to the end of the file already in the buffer. For example, if you type

```
e junk
68          (system response)
r junk
68          (system response)
```

**ed** first places "junk" in the buffer to edit it, and then reads in "junk" again and appends it to the end. You now have two copies of the original text:

```
Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.
```

Like the **w** and **e** commands, **r** prints the number of characters read.

The **r** command can read a file and add it anywhere in the current buffer:

**.r** *filename*

will read the contents of *filename* into the buffer immediately after the current line as opposed to appending it to the end of the file.

The file in the buffer is not destroyed—it continues after the last line of the file you read in. For example, using the original "junk" file:

```
ed
e junk
68                  (system response)
1                   (go to line 1)
Now is the time     (system response)
.r junk
68                  (system response)
w
136                 (system response)
q
```

would place this in your buffer:

```
Now is the time
Now is the time
for all good men
to come to the aid of their party.
for all good men
to come to the aid of their party.
```

The file you read remains in its original file also. You only copied it into the buffer.

EXERCISE 2

Experiment with the **e** command—try reading and printing files. You may get an error like **?** *name* where *name* is the name of a file. This means that the file you referred to does not exist which can be caused by misspelling the file name or trying to read a file you don't have permission to read. Try alternately reading and appending to see that they work similarly.

Verify that

**ed** *filename*

is the same as

**ed**
**e** *filename*





What does

**f** *filename*

do?

### 3.7 Printing Buffer Contents

To print all or part of the buffer on the terminal, use the **p** (print) command. You must specify the line numbers where you want printing to begin and end. These numbers should have a comma between them (*beginning line number*, *ending line number* **p**). For example, to print the first ten lines of the buffer (lines 1 through 10), type:

```
1,10p
```

Suppose you want to print *all* the lines in the buffer. If you knew the exact number of lines in the buffer, such as 30, you could type "**1,30p**." However, if you don't know how many lines there are in the buffer, **ed** has a symbol meaning "line number of the *last line* in the buffer"—the dollar sign, **$**. Therefore, to print all the lines in the buffer, type:

```
1,$p
```

Since this is a common command, "**1,$p**" can be abbreviated to "**$,p**." To stop printing, hit the DEL or DELETE key. **Ed** responds with

```
?
```

and waits for the next command.

To print the *last* line of the buffer, you can type

```
$,$p
```

or

```
$p
```

You can print any single line by typing the line number. For example, typing

```
1
```

prints

Now is the time

which is the first line of the buffer.

You can also use **$** in combinations like

$-5,$p

This example prints the *last five* lines of the buffer.

EXERCISE 3

Create some text with **a** and experiment with the **p** command. You may find that you can't print line 0 or a line beyond the last line of the buffer. **Ed** also won't print the buffer in reverse order. For example,

3,1p

will *not* work.

### 3.8 Current Line

Suppose the buffer contains this text:

Now is the time
for all good men
to come to the aid of their party.
Now is the time
for all good men
to come to the aid of their party.

Typing:

1,3p

will print the first three lines.

Now type:

p

This prints





to come to the aid of their party.

which is the third line of the buffer. It is the last (most recent) line that anything was done to— the line last printed. If you hit **p** again, it will print line 3 again.

The reason for this is that **ed** remembers the last line that anything was done to (in this case, line 3—the last line printed). This most recent line is referred to as

. (pronounced "dot")

Dot is a line number in the same way that **$** is. Dot means "the current line," or "the line something was done to most recently." You can use dot in several ways—one possibility is to enter:

.,$p

This prints everything from the current line to the last line of the buffer. In our example, these are lines 3 through 6.

Some commands change the value of dot, others do not. The **p** command resets dot to the number of the last line printed. For example, **.,$p** sets "." to the last line in the buffer (line 6).

Dot is most useful in combinations like:

.+1 (this is equivalent to .+1p)

This means "print the next line" and is a handy way to step slowly through a buffer. You can also type:

.-1 (or .-1p)

which means "print the line before the current line." This allows you to move backward through the buffer. Another useful example is:

.-3,.-1p

which prints the previous three lines.

Don't forget that all of these commands change the value of dot. You can find out what dot is by typing

```
.=
```

Let's summarize the **p** command and dot. You can precede **p** by 0, 1, or 2 line numbers. If you don't specify a line number, **p** prints the current line (the line that dot refers to). If you specify one line number, (with or without the letter **p**), **ed** prints that line and sets dot there. If you specify two line numbers separated by a comma and followed by **p**, **ed** prints everything from the first number to the last number, and sets dot to the last line printed. (The first number must be smaller than the second number—**ed** won't print backwards.)

Pressing RETURN once prints the next line. It is equivalent to:

```
.+1p
```

Typing ^ or the minus – moves the current line back one line. It can be used in multiples; typing the ^^^ moves the current line back three lines. The "–" and "^" are the same as "–1p."

### 3.9 Deleting Lines

The **d** (delete) command deletes lines. You specify the lines to delete as follows:

*starting line*, *ending line* **d**

This is the same format you just learned for the **p** command. Typing:

```
4,$d
```

deletes everything from line 4 to the end of the buffer. In our example, this deletion leaves us with three lines. We can check these lines by typing:

```
1,$p
```

The last line, $, is now line 3. Dot is set to the line *after* the last line deleted (unless you delete the last line in the buffer). If you delete the last line (as in the last example), dot is set to $.

The **d** (delete) command and the **p** (print) command may be used together. For example, typing:





```
dp
```

deletes the current line, prints the following line, and sets dot to the line printed.

EXERCISE 4

Experiment with **a**, **e**, **r**, **w**, **p**, and **d** until you are familiar with them. Use "dot," $, and line numbers to get familiar with their use.

Try using line numbers with **a**, **r**, and **w**. You will find that **a** appends lines *after* the line number you specify; **r** reads a file in *after* the line number you specify; and **w** writes out the lines you specify. For example, you can insert a file at the beginning of the buffer by typing:

**0r** *filename*

You can enter lines at the beginning of the buffer by typing:

```
0a
text
.
```

You might also notice that

```
.w
```

is *very* different from

```
.
w
```

### 3.10 Changing Text

We are now ready to try one of the most important commands—the **s** (substitute) command.

This command changes words or letters. For example, you will use the substitute command to correct spelling mistakes and typing errors.

Suppose that line 1 is;

Now is th time

You can change "th" to "the" by typing:

1s/th/the/

This says: in line 1, change "th" to "the." Since **ed** doesn't print the change automatically, type

p

to make sure that the substitution worked. You should get

Now is the time

You may have noticed that the substitute command reset the current line (that's why we typed **p** instead of **1p**).

The general format of the substitute command is:

*starting-line*, *ending-line* s/*change this*/*to this*/

The characters between the first and second slashes are replaced by the characters between the second and third slashes. This substitution takes places on *all* lines between *starting-line* and *ending-line*. However, only the *first* occurrence on *each* line is changed. To change *every* occurrence, add **g** (for "global") to the s command, like this:

s/*something*/*something else*/**gp**

The rules for line numbers are the same as those you learned for the print command, **p**. However, if substitute can't find the characters you asked it to change, it doesn't change the current line. **Ed** tells you when this has happened by printing **?** on the screen.

As an example of substitute, you could type

1,$s/speling/spelling/

to correct the first spelling mistake ("speling," in this case) on each





ne. (This is useful for people who make the same mistake consistently.)

f you don't specify a line number, s assumes you want to make the ubstitution on the current line. For example, you could type:

s/*something*/*something else*/**p**

his corrects a mistake on the current line and then prints the current ne to make sure it worked out.

ou can also type:

s/*something*//

his replaces "something" with *nothing*, i.e., removes it. This is useful or deleting extra words in a line or removing extra letters from words.

or example,

Nowxx is the time

an be corrected by typing

s/xx//

he line now reads:

Now is the time

he // (two adjacent slashes) means "no characters," *not* a blank.

## EXERCISE 5

xperiment with the substitute command. For example, type:

a
the other side of the coin
.
s/the/on the/p

This produces the following:

on the other side of the coin

This substitute command changes only the first occurrence. You can change all occurrences by adding **g**.

Try using characters (except blanks and tabs) other than slashes to set off the two sets of characters in the s command. For example, try typing

s'the'other'p

Strange results may be produced by using

^ . $ [ * \ &

Read "Special Characters" in this chapter.

### 3.11 Context Searching

When you master the substitute command, you may want to try another important feature of **ed**—context searching.

Suppose you have these three lines in your buffer:

Now is the time
for all good men
to come to the aid of their party.

If you wanted to locate the word "their," you could type 3. However, if the buffer contained several hundred lines and you had been deleting and rearranging lines, you might have a difficult time locating this line. *Context searching* lets you find a line by specifying some context (unique text) in it.

To search for a line that contains a particular string of characters, type:

*/string of characters to find/*





For example,

/their/

locates the next occurrence of "their." It also makes that line the current line and prints it for verification.

"Next occurrence" means **ed** starts looking for the string at line after the current line (.+1) and searches to the end of the buffer. Then it searches from line 1 to the line that it started searching at (dot). That is, the search "wraps around" from **$** to **1**. It scans all the lines in the buffer until it either finds the desired string or gets back to dot again. If **ed** can't find the characters, it types the error message

?

Otherwise, it prints the line it found.

You can search for the desired line *and* make a substitution to it in the same command, like this:

/their/s/their/the/p

This tells **ed** to: search for the word "their," substitute "the" for "their," and print the new line. When it has finished, **ed** prints this:

to come to the aid of the party.

You can repeat a context search. For example,

/string/

finds the next occurrence of "string." If this is not the line you want, you can search for the next occurrence by typing:

//

This stands for "the previous context search expression." This abbreviation can also be used as the first string of the substitute command. For example:

/string1/s//string2/

finds the next occurrence of "string1" and replaces it with "string2." Similarly,

```
??
```

scans backwards for the previous expression.

Context searches (like "/their/") are interchangeable with line numbers. They can be used alone to find a desired line or as line numbers for some other command—like s.

Suppose the buffer contains these three familiar lines:

```
Now is the time
for all good men
to come to the aid of their party.
```

The following context search expressions:

```
/Now/+1
/good/
/party/-1
```

all refer to the same line (line 2). To make a change in line 2, you can type:

```
/Now/+1s/good/bad/
```

or

```
/good/s/good/bad/
```

or

```
/party/-1s/good/bad/
```

You could print all three lines by typing either:

```
/Now/,/party/p
```

or

```
/Now/,/Now/+2p
```

The first of these might be better if you don't know how many lines there are. A context search expression is the *same* as a line number, so it can be used wherever you would use a line number.





EXERCISE 6

Experiment with context searching. Try using context searches as line numbers for the substitute, print, and delete commands. They can also be used with **r**, **w**, and **a**.

Try context searching using "?text?" instead of "/text/" This searches backwards for the character string.

If you have any problems with the characters

```
^  .  $  [  *  \  &
```

read the section on "Special Characters."

### 3.12 Change and Insert Commands

In this section we discuss the **c** (change) command—which changes the current line or replaces the current line with one or more lines; and the **i** (insert) command—which inserts one or more lines before the current line.

For example, to replace everything past the current line to the last line, type:

```
.+1,$c
one or more lines of text
.
```

The text typed after the **c** command to the "." command will take the place of the original text between the ".+1" line and the last line. This command is useful when you want to replace a line or several lines which have errors.

If you specify only one line, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice that you end your changes by typing a period (.) at the beginning of a line—this is the same way you stopped adding text with the **a** (append) command. If you don't specify a line number, **c** replaces the current line. When you finish making changes, dot is set to the last line you inserted.

Insert is similar to append except that it inserts the text *before* rather than *after* the current or specified line—for example, typing:

```
2i
one or more lines of text
.
```

inserts the text *before* the second line. If you don't specify a line number, the text is inserted before the current line. Dot is set to the last line inserted.

EXERCISE 7

A change is like a combination of a delete followed by an insert. Experiment to verify that:

*starting-line*,*ending-line*d
i
*text*
.

is almost the same as

*starting-line*,*ending-line*c
*text*
.

These are not *exactly* the same if the last line is deleted. What is dot in this case?

Experiment with the append command a and the insert command i to see that they are similar but not the same. You will notice that

*line-number*a
*text*
.

appends *after* the given line, while





*line-number*i
*text*
.

inserts *before* it. If you don't specify a line number, i inserts before the current line, a appends after the current line, and c changes the current line.

### 3.13 Moving Text

The m (move) command moves lines from one place to another. Suppose you want to move the first three lines of the buffer to the end. You could type:

```
1,3w filename
$r filename
1,3d
```

This would work, but the following is easier:

```
1,3m$
```

The general case is:

*starting-line*,*ending-line*m *after-this-line*

The text is moved after the specified line number. You can use context searches instead of line numbers. For example, if in your buffer you had:

```
First paragraph
...
end of first paragraph.
Second paragraph
...
end of second paragraph.
```

you could reverse the two by typing:

```
/Second/,/end of second/m/First/-1
```

The "−1" was used because the text is moved *after* the line specified. Dot is set to the last line moved.

## 4. Global Commands

The two global commands are **g** and **v**. G executes commands on all lines that match some specified string. For example

```
g/speling/p
```

prints all lines that contain "speling."

```
g/speling/s//spelling/gp
```

replaces "speling" with "spelling" each time it occurs (even if it occurs more than once in a line), then prints each corrected line.

Compare this to

```
1,$s/speling/spelling/gp
```

This only prints the last line substituted. G doesn't print ? if "speling" is not found.

You can use several commands at a time with **g**. Just remember to end every line but the last with a backslash "\". For example:

```
g/xxx/-1s/abc/def/\
.+2s/ghi/jkl/\
.-2,.p
```

makes changes in the lines before and after each line containing "xxx," then prints all three lines.

The **v** command is the same as **g** except that it executes the commands on lines that *don't* match the string. For example:

```
v/ /d
```

deletes every line that does not contain a blank.

## 5. Special Characters

You may have noticed that some characters (such as ., *, $) changed the meaning of context searches and the **s** command. This is because these characters have special meanings for **ed**.





The following is a complete list of the special characters that can cause trouble:

```
.  ^  $  *  [ ]  &  \
```

### 5.1 Period

The "." (period) signifies *any character*. In a context search or the first string of the substitute command,

```
/x.y/
```

means

```
/x any-character y/
```

This example will find all the following:

```
x+y
x-y
x y
x.y
xAy
```

### 5.2 Circumflex

The circumflex "^" signifies the beginning of a line. For example:

```
/^string/
```

finds "string" only if it is at the beginning of a line. That is, it will find

```
string
```

but not

```
the string
```

### 5.3 Dollar Sign

The dollar sign "$" is just the opposite of the circumflex; it means the end of a line.

Typing:

```
/string$/
```

finds "string" only at the end of some line.

/^string$/

will find a line containing only "string" and

/^.$/

finds a line containing one character.

### 5.4 Asterisk

The asterisk (*) is the repetition character. Therefore, "a*" means "any number of a's." ".*" matches any number of anythings.

For example,

s/.*/stuff/

changes an entire line to "stuff," and

s/.*,//

deletes all the characters in a line up to and including the last comma. (Since ".*" finds the longest possible match, this matches the last comma rather than the first.)

### 5.5 Brackets

The left bracket ([) is used with right bracket (]) to enclose *character classes*. For example,

/[0123456789]/

matches any single digit. This can be abbreviated as

[0-9]

It can also represent the alphabet:

[A-Z]

or

[a-z]

### 5.6 Ampersand

Finally, the ampersand (&) (used only on the right-hand part of a substitute command) means "whatever was matched on the left-hand side."





Suppose the current line contained

Now is the time

and you wanted to put parentheses around it. You could input:

s/^/(/
s/$/)/

However, an easier way to accomplish the same thing is to type:

s/.*/(&)/

This says "match the whole line and replace it by itself surrounded by parentheses." The "&" can be used several times in a line. Using the above example text:

s/.*/&? &!!/

produces:

Now is the time? Now is the time!!

You don't have to match the whole line. If the buffer contains

the end of the world

you could type:

/world/s//& is at hand/

to produce:

the end of the world is at hand

the sequence "/world/" found the desired line; the sequence "//" found the same word in the line; and the "&" saved you from typing world again.

The "&" is a special character only in the replacement text of a substitute command.

### 5.7 Backslash

If you have to use one of the special characters without its special meaning in a substitute command, precede it with the backslash. For example:

```
s/\./dot/
```

changes the first occurrence of a "." into the word "dot" on the current line. If the "." was not preceded by the "\," the result would have been that the first character would have been changed into the word "dot" on the current line.

## 6. Summary of Commands

The general form of **ed** commands is **name**, which may be preceded by one or two line numbers. The edit command **e**, the read command **r**, and the write command **w**, are also followed by a "file name." Most commands are used separately.

**a** — **append** adds text after the current line, or after the line number specified. To stop adding text, type a period (.) at the beginning of a line and hit RETURN. Dot is set to the last line appended.

**c** — change the specified lines to the new text which follows. To stop replacing text, type a period (.) at the beginning of a line and hit RETURN. If you don't specify a line, the current line is replaced. Dot is set to last line changed.

**d** — delete the specified lines. If you don't specify a line, the current line is deleted. Dot is set to the line after the last deleted line. If you delete the last line in the buffer, dot is set to the new last line.

**e** — edit new file. The previous contents of the buffer are destroyed, so save your work before you read in a new file with **e**.

**f** — Print the current file name. This is the file **ed** assumes you mean if you don't specify a file. To change the current file name, type **f** *filename*.

**g** — The global command "**g**/---/*commands*" executes the commands on lines containing "---".

**i** — insert lines before the specified line or the current line. To stop inserting text, type a period (.) at the beginning of a line and hit RETURN. Dot is set to last line inserted.





**m** — move the text (between the first two specified lines) after the third line specified. Dot is set to the last line moved.

**n** — Print the number of the addressed line(s), a tab, and the line itself.

**p** — print the lines specified. If you don't specify any line number, **p** prints the current line. Pressing RETURN prints the next line.

**q** — The quit command exits **ed**. You will most likely want to write your text before using quit. However, if you don't want to save your changes, hit **q** twice.

**r** — read a file into the end of the buffer. "**.r** *filename*" reads the file into the buffer after the current line. Dot is set to the last line read.

**s** — substitute "string2" for "string1" in the specified lines (**s**/*string1*/*string2*/). If you don't specify any lines, it makes the substitution in the current line. The **s** changes only the first occurrence of "string1" on a line. To change all occurrences, type **g** at the end of the command. Dot is set to last line in which a substitution took place; if no substitution took place, dot is not changed.

**v** — The exclude command (**v**/---/*commands*) executes commands only on lines *not* containing "---".

**w** — The write command writes the buffer onto a file. Dot is not changed.

**.=** — The ".=" string prints the current line number. The "=" by itself prints the line number of the last line in the file.

**!** — The "!" is a *temporary escape* command. Typing "!*command-line*" within **ed** executes "*command-line*" and then returns you to the editor.

**/-----/** — The *context search* command searches for next line containing "----" and prints it. The search starts at the line after the current line, reads to the end of the buffer, then wraps around to line 1 and searches to the original line. Dot is set to the line where the string is found.

?-----?  Performs a backwards *context search*. It begins searching at the line before the current line, reads backwards to the start of the file, then wraps around to the end of the file and searches backwards to the original line. Dot is set to the line where the string is found.

?-----?  Performs a backwards *context search*. It begins searching at the line before the current line, reads backwards to the start of the file, then wraps around to the end of the file and searches backwards to the original line. Dot is set to the line where the string is found.



# Chapter 4: VISUAL TEXT EDITOR — VI

CONTENTS

# Chapter 4

# VISUAL TEXT EDITOR — VI

The visual editor **vi** is a screen-oriented, interactive text editor. It uses your screen as a window into the file you are editing. When you make a change, you see it on your screen.

**Vi** inserts new text easily. Most of the **vi** commands move the cursor around the file. For example, there are commands that move the cursor forward and backward over characters, words, sentences, and paragraphs. You can combine motion commands with some operators (such as **d** for delete and **c** for change) to form larger operations (such as delete word or change paragraph). This regularity and the mnemonic assignment of commands makes **vi** commands easy to remember and to use.

The **vi** commands are available on all terminals. You can also use commands for **ed** (the line-oriented editor) within **vi**; it is easy to switch between these two editors.

## 1. Getting Started

This chapter introduces **vi** (pronounced *vee-eye*). The first part of this chapter (sections 1 through 5) describes the basic uses of **vi**. We present some special-interest topics in section 6, and some nitty-gritty details in section 7.

There is also an appendix listing the special meanings of some characters in **vi**.

### 1.1 Editing a File

Make a copy of an existing file called "temp" to use for practicing text editing with **vi**. Start to edit this file by typing:

% vi tempRETURN

where "temp" is the name of the file copy you just created. The screen clears and the text of your file appears on the screen. If something else happens, refer to the footnote.*

where "temp" is the name of the file copy you just created. The screen clears and the text of your file appears on the screen. If something else happens, refer to the footnote.*

### 1.2 Visual Editor's Copy: The Buffer

The visual editor does not directly modify the file you edit. Rather, it makes a copy of this file and places it in the *buffer*. You do not change the contents of the file until you write the changes you make back into the original file.

### 1.3 Arrow Keys

The arrow keys on your keyboard move the cursor in **vi**. You can also use the **h j k** and **l** keys to move the cursor:

- **h** moves a space to the left (the same as a backspace),
- **j** moves down a line (in the same column),
- **k** moves up a line (in the same column), and
- **l** moves a space to the right.

### 1.4 Special Characters: ESC, RETURN and DEL

Look on your keyboard for a key labelled ESC or ALT. It is probably near the upper-left corner of your keyboard. Try pressing this key a few times. On most terminals a bell will ring. This means that the visual editor does not understand why you pressed this key.† You will

* If there is no text on your screen, you probably typed the wrong file name. If typed the incorrect file name, **vi** creates a new file with the name that you input. To get out of the file type "**:q**" and try again. You might also have copied an empty file—in this case, quit the file by typing "**:q**" and remove the file you created by typing "**rm temp**". Try copying another file.

† On some terminals the visual editor will quietly flash the screen rather than ringing the bell.





be using the ESC key to cancel commands and to signal that you are through inserting text.

The RETURN or CR (carriage return) key terminates certain commands. It is usually at the right side of the keyboard and is the same key you press at the end of each shell command. We indicate pressing this key in our examples with RETURN.

Another very useful key is the DEL (possibly DELETE or RUB on some keyboards) key, which tells **vi** to stop what it is doing. It is a forceful way of making **vi** listen to you, or to stop it from doing something you don't want it to do.

Try pressing the "/" key. Pressing this key means you want to search for some text. The cursor should now be at the bottom line of the terminal. You can get the cursor back to the current position by hitting the DEL key; try this now.‡ From now on we will simply refer to pressing the DEL key as "sending an interrupt."

**Vi** often shows your commands on the last line of the terminal. If the cursor is on the first position of this last line, the visual editor is working on something (such as finding a new position in the file after a search or reformatting the buffer). When this happens, you can stop **vi** by sending an interrupt.

### 1.5 Getting Out of the Editor

After you have worked with this introduction for a while, you can give the command **ZZ** or **:wq**. This will write the contents of the buffer back into the file you are editing, and then quit (leave) the visual editor. You can also quit an editing session by giving the command

:q!RETURN

this is a dangerous but occasionally essential command which ends the session and discards all your changes. You will use this command if you make major changes you don't want to keep, such as erasing half

‡ Backspacing over the "/" will also cancel the search.

of your file by mistake. Don't use this command if you want to save your changes.

## 2. Moving Around in the File

### 2.1 Scrolling and Paging

The visual editor has several commands that move you around your file. The most useful of these is CONTROL-D (this is produced by holding down the key labelled CONTROL or CTRL while you press the "d" key). We will use this notation for referring to these *control* keys from now on.

A CONTROL-D scrolls half a screenful (12 lines) down in the file. The "d" stands for *down*. Many vi commands are mnemonic and this makes them much easier to remember. For instance, the command to scroll *up* is CONTROL-U.

If you want to see more of the file below where you are, you can press CONTROL-E to *expose* one more line at the bottom of the screen, leaving the cursor where it is. The command CONTROL-Y (which is hopelessly non-mnemonic, but next to CONTROL-U on the keyboard) exposes one more line at the top of the screen.

There are other ways to move around in a file; the keys CONTROL-B and CONTROL-F move backward and forward a page (a screenful), respectively, keeping a couple of lines of continuity between screens.

Notice the difference between scrolling and paging. If you are trying to read through a file, pressing CONTROL-F to move forward a page will leave you only a little context to look back at. Scrolling, on the other hand, leaves more context. Sometimes you can continue to read the text as scrolling takes place.

### 2.2 Searching, Goto, and Previous Context

You can also move around by searching for text that appears at the location you want to go to. To do this you would type "/" and a string of characters to search for, then press RETURN. For example, type:

```
/theRETURN
```





Vi moves the cursor to the next occurrence of this string. Press n to search for the *next* occurrence of this string. Pressing ? searches backwards for the string.

If the visual editor cannot find the string you're searching for, it will print a message on the last line of the screen and return the cursor to its initial position.

If you want to search for a string that only appears at the beginning of a line, type ^ before the string. For example, typing:

```
/^theRETURN
```

searches for the incidence of the word "the" only when it appears at the beginning of a line. To search for a string that appears only at the end of a line, end a search string with a $. Thus:

```
/the$RETURN
```

searches for the incidence of the word "the" only when it appears at the end of a line.

Typing a line number and G moves your cursor to that line in the file. For example,

```
1GRETURN
```

moves the cursor to the first line of the file. If you type G with no line number, the visual editor moves your cursor to the end of the file.

Move to the end of your file by hitting G. Notice that vi places only the tilde character (~) on each line past the last line of your text. This means that the lines marked with "~" are past the end of the file.

You can find out information about the file you are editing by typing CONTROL-G. The last line of your screen will show you the name of the file you are editing, the current line number, the total number of lines in the buffer, and the percentage of the way through the buffer you are.

You can get back to a previous position by using the two back quotes (``). This is often more convenient than G because frequently you

don't remember the previous line number. Move to line 5 in your file by typing "5G". Now type G which will bring you to the end of your file, then type `` to get back to where you were (line 5). If you accidentally hit another command which moves you away from where you wanted to be, you can quickly get back by pressing ``.

### 2.3 Moving Around on the Screen

Now move the cursor around on the screen. If your terminal has arrow keys use them. Also you can always use the keyboard letters **h** to move left, **j** to move down, **k** to move up, and **l** to move right. Experienced users of **vi** prefer these keys to arrow keys because they are usually right underneath their fingers.

Press the + key. Each time you do, the cursor moves to the next line in the file. The RETURN key has the same effect as the + key. The − key moves you back a line.

These are common keys for moving up and down lines in the file. If you go off the bottom or top of the screen with these keys, the screen scrolls down or up to bring one line at a time into view.

**Vi** also has commands to take you to the top, middle, and bottom of the screen:

| | |
|---|---|
| **H** | top |
| **M** | middle |
| **L** | last |

**H** takes you to the top (*home*) line on the screen. Try preceding it with a number, for example, "3H". This takes you to the third line on the screen. **M** takes you to the *middle* line on the screen. **L** takes you to the *last* line on the screen. You can also precede **L** with a number. For example, "5L" takes you to the fifth line from the bottom.

### 2.4 Moving Within a Line

Pick out a word somewhere in the middle of the screen. Move the cursor (using RETURN or −) to the line where the word is. Press the **w** key. This advances the cursor to the start of the next *word* on the line. Press the **b** key to *back* up words in the line. Also press the **e** key to move to the *end* of the current word rather than to the beginning of the





next word. Hitting the space bar moves right one character and the BACKSPACE key key moves left one character. (Also, as we mentioned before, **h** moves you to the left and **l** moves you to the right.)

If you move on a line with punctuation in it, you may notice the **w** and **b** keys stop at each punctuation mark. You can go forward and backward without stopping at punctuation by using **W** and **B**, respectively, rather than the lowercase equivalents. Try these on a few lines with punctuation to see how they differ from the lowercase **w** and **b**.

The word keys wrap around the end of line, rather than stopping at the end. Move to a word on the line below by repeatedly pressing **w**.

### 2.5 Summary

| | |
|---|---|
| **CONTROL-B** | move backwards to previous page |
| **CONTROL-D** | scroll down in the file |
| **CONTROL-E** | expose another line at the bottom |
| **CONTROL-F** | move forward to next page |
| **CONTROL-G** | tell what is going on |
| **CONTROL-H** | move the cursor back a space |
| **CONTROL-N** | move to next line, same column |
| **CONTROL-P** | move to previous line, same column |
| **CONTROL-U** | scroll up in the file |
| **CONTROL-Y** | expose another line at the top |
| **+** | move to next line, at the beginning |
| **−** | move to previous line, at the beginning |
| **/** | search forward for the following string |
| **?** | search backward for the following string |
| **B** | move back a word, ignoring punctuation |
| **G** | go to specified line, default is last line |
| **H** | go to home screen line |
| **M** | go to middle screen line |
| **L** | go to last screen line |
| **W** | move forward a word, ignoring punctuation |
| **b** | move back a word |
| **e** | move to end of current word |
| **n** | search for next instance of / or ? pattern |
| **w** | move to word after this word |

### 2.6 Viewing a File

If you want to look at a file, rather than make changes, start the visual editor by typing:

view *filename*

instead of **vi**. This prevents you from accidently overwriting or changing the file.

## 3. Making Simple Changes

### 3.1 Inserting

One of the most useful commands is the **i** (insert) command. After you type **I**, everything you type is inserted into the file. To finish inserting, press the ESC key. Try this now; move to the beginning of a word in your file and insert text before this word by and pressing **i** and typing in some words and then ESC. If you are on an dumb terminal it will seem, for a minute, that some of the characters in your line have been overwritten, but they will reappear when you press ESC.

Notice that **I** inserts what you have typed before the cursor position. The **a** command *appends* text after the cursor position. Move to the start of a word that can be pluralized and type **e** (to move to end of word), then **a** (to append) and type the letter "s" and ESC. This sequence of commands pluralizes a word.

Insert and append a few times to make sure you understand how this works; **I** places text to the left of the cursor, **a** to the right.

You will often want to add new lines to the file you are editing. Press **o** which creates (*opens*) a new line after the line you are on. Use the **O** command to create a new line before the line you are on. After you create a new line in this way, everything you type is inserted on the new line. To stop inserting, press ESC.

Many related editor commands are started by the same letter but one is lowercase and the other is uppercase. The uppercase key often works backward and/or up, while the lowercase key moves forward and/or down.





To type in more than one line of text, press RETURN where you want to end a line. This creates a new line for text and you can continue typing. If you are on a slow or dumb terminal, the visual editor may wait to redraw the screen, and will let you type over the existing screen lines. This avoids waiting for the editor to keep the end of the screen up to date. The end of the screen will be fixed, and the missing lines will reappear when you press ESC.

While you are in *insert mode* (inserting new text using), you can press **CONTROL-H** or **#** to backspace over the last character you typed; and **@**, **CONTROL-X**, or **CONTROL-U** to erase the current line you typed. **CONTROL-W** erases a whole word and leaves you after the previous word; it is useful for quickly erasing an insert.

When you backspace during an insertion, the characters you backspace over remain on your screen. This is useful if you want to type in something similar. The characters disappear when when you press ESC.

When you are in insert mode you cannot erase characters which you didn't just insert, and you cannot backspace up to the previous line. If you need to back up to the previous line to make a correction, press ESC and move the cursor back to the previous line. After making the correction (see next section), you can return to where you were and use the insert or append command again.

### 3.2 Making Small Corrections

You can make small corrections in existing text quite easily. Find a single character to correct. Use the arrow keys or word motion keys to move the cursor to the character. Press the **x** key to delete the character. This is analogous to the way you "x" out characters when you make mistakes on a typewriter (except it's not as messy).

If the character is incorrect, you can replace it with the correct character by typing **r***c*, where *c* is the correct character. Also, if you want to replace a character with more than one character,

**s***string*ESC

makes the substitution. If there are a few characters which are wrong, you can precede **s** with the number of characters to replace. Counts are also useful with **x** to specify the number of characters to delete.

### 3.3 More Corrections: Operators

You already know almost enough to make changes at a higher level. All you need to know now is that the operator **d** deletes. Move to the beginning of a word and type **dw** to delete a word.

Try hitting the . a few times. Notice that this repeats the **dw** command. The command . repeats the last command which made a change. You can remember it by analogy with an ellipsis "...".

Now try **db**. This deletes a word backwards. Try **d** and a space; this deletes a single character and is equivalent to the **x** command.

Another very useful operator is **c** for change. The command **cw** changes a single word. You need to follow the command with a replacement word or words, and then press ESC. Move to the beginning of a word and type:

**cw** *newword*ESC

Notice that the end of the text to be changed was marked with the character "$".

### 3.4 Operating on Lines

The command **dd** deletes a line. If you are on a dumb terminal, **vi** may just erase the line on the screen, replacing it with a line with only an "@" symbol on it. This line does not correspond to any line in your file, but acts as a place holder. It helps to avoid a lengthy redraw of the rest of the screen.

The command **cc** changes a whole line, erasing its previous contents and replacing them with text you type until you press the ESC.*

* The command S is the same as cc. Think of S as a substitute of lines, while s is a substitute on characters.





You can delete or change more than one line by preceding the **dd** or **cc** with a number. For example, "**5dd**" deletes 5 lines. You can also give a command like "**dL**" to delete all the lines up to and including the last line on the screen, or "**d3L**" to delete through the third line from the bottom. Try some commands like this now.

One way to use these operators involves using the / search command. For example:

**d/***pattern*

will delete all characters from the current position to the *pattern*.

**Vi** lets you know when you change many lines by broadcasting the message on the last line of the screen. The visual editor also tells you when a change affects text which you cannot see.

### 3.5 Undoing

Suppose that the last change you made was incorrect; the visual editor has a **u** (undo) command to reverse the last change you made. Try this a few times. Notice that **u** also undoes a **u**.

The undo command reverses only a single change. After you make several changes to a line, you may decide that you would rather have the original line back. The **U** command restores the current line to the way it was before you started changing it.

You can recover text which you delete, even if undo will not bring it back; see the section on recovering lost text below.

### 3.6 Summary

| | |
|---|---|
| CONTROL-W | erase a word during insert mode |
| CONTROL-H | erase a character during insert mode |
| CONTROL-X | kill the insert on this line |
| CONTROL-U | kill the insert on this line |
| . | repeat the last changing command |
| O | open and input new lines, above the current line |
| U | undo the changes made to the current line |
| a | append text after the cursor |
| c | change the object you specify to the following text |
| d | delete the object you specify |
| i | insert text before the cursor |
| o | open and input new lines, below the current |
| u | undo the last change |

## 4. Moving About; Rearranging and Duplicating Text

### 4.1 Working with Characters

Move the cursor to a line with a parenthesis, comma, or period. Try the command:

**f***x*

where *x* is the punctuation mark. This command *finds* the next *x* character to the right of the cursor in the current line. Try pressing ; (semi-colon). This finds the next occurrence of the same character. There is also a **F** command, which is like **f**, but searches backward. The ; command repeats **F** also.

Type

**df***x*

where *x* is a character on the current line. Notice that text up to and including the *x* character is deleted. Undo this with **u** and then try

**dt***x*

where *x* is a character on the current line. This command deletes up *to* but not including the *x*. The **t** command stands for to. The command T is the reverse of **t** (it works backwards).





The ^ character moves the cursor to the first non-white position on the line, and a **$** moves it to the end of the line. Thus **$a** appends new text to the end of the current line.

Your file may have tab (CONTROL-I) characters in it. These characters are represented as several spaces expanding to a tab stop, where tab stops are every 8 positions. When the cursor is at a tab, it sits on the last of the spaces which represent that tab. Try moving the cursor back and forth over tabs so you understand how this works.

On rare occasions, your file may have nonprinting characters in it. These characters are displayed as a two-character code, the first character of which is "^". On the screen, non-printing characters resemble a "^" character adjacent to another, but spacing or backspacing over the character reveals that the two characters are treated as a single character.

The visual editor sometimes discards control characters (depending on the character and the setting of the *beautify* option) if you attempt to insert them in your file. You can get a control character in the file by beginning an insert and then typing a CONTROL-V before the control character. The CONTROL-V quotes the following character, inserting it directly into the file.

### 4.2 Working on Sentences, Paragraphs, and Sections

You may want to use commands on whole sentences, paragraphs, and sections. The ( and ) move to the beginning of the previous and next sentences, respectively. Thus the command "**d)**" will delete the rest of the current sentence; likewise "**d(**" will delete the previous sentence if you are at the beginning of the current sentence, or the current sentence up to where you are if you are not at the beginning of the current sentence.

A sentence ends at a . ! or ? which is followed by either the end of a line or by two spaces. Any number of closing ) ] " and ' characters may appear after the . ! or ?.

The { and } move over paragraphs and || and || move over sections.

A paragraph begins after each empty line, and also at a paragraph macro. The default setting for this option defines the paragraph macros of the **−ms** and **−mm** macro packages, i.e., the **.IP**, **.LP**, **.PP** and **.QP**, **.P** and **.LI** macros.† The **troff** request **.bp** also starts a paragraph. Each paragraph boundary is also a sentence boundary. The sentence and paragraph commands can be given counts to operate over groups of sentences and paragraphs.

Sections begin after each macro in the *sections* option (normally **.NH**, **.SH**, **.H** and **.HU**) and each line with a formfeed (**CONTROL-L**) in the first column. Section boundaries are always line and paragraph boundaries.

Try experimenting with the sentence and paragraph commands until you are sure how they work. If you have a large document, try looking through it using the section commands. The section commands interpret a preceding count as a different window size for newly drawn windows until you specify another size. This is very useful if you are on a slow terminal and are looking for a particular section. You can give the first section command a small count to then see each successive section heading in a small window.

### 4.3 Rearranging and Duplicating Text

**Vi** has a single unnamed buffer where the last deleted or changed text is saved, and a set of named buffers (a through z) which you can use to save copies of text and to move text around in your file and between files.

The operator **y** yanks a copy of the object which follows into the unnamed buffer. For example, pressing "**y3w**" puts three words in the buffer. If preceded by a buffer name, such as:

"*x*y

where *x* is replaced by a letter a-z, it places the text in the named buffer. The text can then be put back in the file with the commands **p** and **P**; **p** *puts* the text after or below the cursor, while **P** *puts* the text before or above the cursor.

If the text you yank forms part of a line, or partially spans more than one line, when you put the text back, it will be placed after the cursor (or before, if you use **P**). If the yanked text forms whole lines, they will be put back as whole lines, without changing the current line. In this case, the put acts much like a **o** or **O** command.

Try the command **YP**. This makes a copy of the current line and places it before the current line. The command **Y** is a convenient abbreviation for **yy**. The command **Yp** also makes a copy of the current line and places it after the current line. You can give **Y** a count of lines to yank and duplicate several lines. Try typing "**3YP**".

To move text within the buffer, you need to delete it in one place and put it back in another. You can precede a delete command by the name of a buffer to store the text in. For example:

"a5dd

deletes 5 lines and places them in the buffer "a". You can then move the cursor to where you want the lines moved to and type

"ap

or

"aP

to put them back. An ordinary delete command saves the text in the unnamed buffer, so an ordinary put can move it elsewhere. However, the unnamed buffer is lost when you make any changes,

† You can easily change or extend this set of macros by assigning a different string to the *paragraphs* option in your EXINIT.

### 4.4 Summary

| | |
|---|---|
| ^ | first non-white on line |
| $ | end of line |
| ) | forward sentence |
| } | forward paragraph |
| ]] | forward section |
| ( | backward sentence |
| { | backward paragraph |
| [[ | backward section |
| f*x* | find *x* forward in line |
| t*x* | to *x* forward in line |
| p | put text back, after cursor or below current line |
| y | yank operator, for copies and moves |
| t*x* | up to *x* forward, for operators |
| F*x* | find *x* backward in line |
| P | put text back, before cursor or above current line |
| T*x* | to *x* backward in line |

## 5. High-Level Commands

### 5.1 Writing, Quitting, Editing New Files

So far we have seen how to enter vi and write our file using either ZZ or :w. The first exits from the visual editor (writing any changes made), the second writes and stays in the visual editor.

If you have made changes in the file but do not want to save your changes, type ":q!RETURN" to leave vi without writing the changes. You can also re-edit the same file (starting over) by giving the command ":e!RETURN". These commands should be used with caution, since you cannot retrieve your changes once you've used either of these commands.

You can edit a different file without leaving the visual editor by typing:

:e *name*RETURN

If you have not written your file before you do this, vi will tell you so. You can then type ":wRETURN" to save your work and then retype the above ":e" command again.





### 5.2 Escaping to the Shell

You can get to the shell to execute a single command by giving a vi command of the form:

:!*cmd*RETURN

The system will run the single command *cmd* and ask you to press RETURN when the shell command is finished. When you have finished looking at the output on the screen, press RETURN and the visual editor clears the screen and redraws it. You can then continue editing. You can also remain in the shell by typing : instead of RETURN; in this case, the screen will not be redrawn.

To execute more than one command in the shell, type:

:shRETURN

This gives you a new shell. When you finish with the shell, type CONTROL-D. The visual editor clears the screen and continues.

### 5.3 Marking and Returning

The command `` returned to the previous place after moving the cursor with commands such as /, ? or G. You can also mark lines, using the m command, in the file with single letter tags and return to these marks later by typing the tags. Try marking the current line with the command:

m*x*

where *x* stands for a letter. Then move the cursor to a different line (any way you like) and press:

`*x*

where *x* stands for the letter you choose in the above example. The cursor returns to the place you marked. Marks last only until you edit another file.

When using operators such as d and referring to marked lines, you may want to delete whole lines rather than deleting to the exact position in the line marked by m. In this case you can use

'*x*

rather than

`*x*

Used without an operator, "'*x*" moves to the first non-white character of the marked line; similarly '' moves to the first non-white character of the line containing the previous context mark ''.

### 5.4 Adjusting the Screen

To clean up your screen, press CONTROL-L, the ASCII form-feed character.

On a dumb terminal, if there are "@" lines in the middle of the screen as a result of line deletion, you may get rid of these lines by typing CONTROL-R.

You can redraw your screen to that a certain line will be placed at the top, middle, or bottom of your screen. To do this move the cursor to that line and type **z**. You should follow the **z** command with a RETURN if you want the line to appear at the top of the window, a . if you want it at the center, or a – if you want it at the bottom.

## 6. Special Topics

### 6.1 Editing on Slow Terminals

When you are on a slow terminal, you should limit the output to your screen. We have already shown how **vi** optimizes screen updating during insertions to limit the delays, and how the visual editor erases lines to "@" when they are deleted on dumb terminals.

If you have a slow terminal, you should set the *slowopen* option. You can force the visual editor to use this mode even on faster terminals by giving the command:

:se slowRETURN

If your system is sluggish, this helps lessen the amount of output coming to your terminal. You can disable this option by:





:se noslowRETURN

A dumb terminal can can simulate an intelligent terminal. Try giving the command

:se redrawRETURN

This generates a great deal of output and is generally tolerable only on lightly loaded systems and fast terminals. You can disable this by giving the command:

:se noredrawRETURN

**Vi** also makes editing more pleasant at low speed by starting editing in a small window, and letting the window expand as you edit. This works particularly well on intelligent terminals. The window easily expands when you insert in the middle of the screen on these terminals. If possible, try editing on an intelligent terminal to see how it works.

You can control the size of the window redrawn when the screen is cleared by giving window sizes as argument to the commands:

: / ? [[ ]] ` '

If you are searching for a common string in a file, you can precede the first search command by a small number (for instance, 3) and **vi** will draw three line windows around each instance of the string.

You can easily expand or contract the window, placing the current line where you want, by giving a number after the **z** command and before the following RETURN, . or –. For example, the command:

z5.

redraws the screen with the current line in the center of a five line window.‡

‡ The command "5z." has an entirely different effect, placing line 5 in the center of a new window.

If the visual editor is updating large portions of the display, you can interrupt by pressing DEL. This may partially confuse **vi** about what is displayed on the screen. You can still edit the text on the screen if you wish; clear up the confusion by pressing **CONTROL-L**; or move or search again, ignoring the current state of the display.

See the section on *open* mode for another way to use the **vi** command set on slow terminals.

### 6.2 Options, Set, and Editor Startup Files

**Vi** has a set of options, some of which have been mentioned above. The most useful options are the following:

| Name | Default | Description |
|---|---|---|
| autoindent | noai | Supply indentation automatically |
| autowrite | noaw | Automatic write before :n, :ta, ^], ! |
| ignorecase | noic | Ignore case in searching |
| list | nolist | Tabs print as CONTROL-I; end-of-lines as $ |
| magic | nomagic | The characters . [ and * are special in scans |
| number | nonu | Lines are displayed prefixed with line numbers |
| paragraphs | para=IPLPPP QPbpP LI | Macro names which start paragraphs |
| redraw | nore | Simulate a smart terminal on a dumb one |
| sections | sect= NHSHH HU | Macro names which start new sections |
| shiftwidth | sw=8 | Shift distance for <, > and input CONTROL-D and CONTROL-T |
| showmatch | nosm | Show matching ( or { as ) or } is typed |
| slowopen | slow | Postpone display updates during inserts |
| term | dumb | The kind of terminal you are using. |

There are three kinds of options: numeric options, string options, and toggle options. You can set numeric and string options by a statement of the form

set *opt*=*val*

and toggle options can be set or unset by statements of one of the forms





set *opt*
set no*opt*

These statements can be placed in your EXINIT in your environment, or given while you are running **vi** by preceding them with a : and following them with a RETURN.

You can get a list of all options you have changed by typing

:setRETURN

or the value of a single option by typing

:set *opt*?RETURN

You can list all possible options and their values by typing

:set allRETURN

The **set** command can be abbreviated to se. Multiple options can be placed on one line, e.g.,

:se ai aw nuRETURN

Options set by the **set** command only last while you stay in **vi**. If you want certain options set whenever you edit, create a list of editor commands to run every time you start up *ex*, *ed*, or **vi**. A typical list includes a **set** command and a few **map** commands (we discuss **map** commands later in this chapter). You can fit these commands on one line by separating them with the | character, for example:

set ai aw terse|map @ dd|map # x

which sets the options *autoindent*, *autowrite*, *terse*, (using the **set** command), makes @ delete a line, (the first **map**), and makes # delete a character, (the second **map**). This string should be placed in the variable EXINIT in your environment. If you use csh, put this line in the file ".login" in your home directory:

setenv EXINIT 'set ai aw terse|map @ dd|map # x'

Of course, the particulars of the line would depend on which options you wanted to set.

### 6.3 Recovering Lost Lines

You might have a serious problem if you delete lines and then regret that they were deleted. Despair not, the visual editor saves the last 9 deleted blocks of text in a set of numbered registers 1–9. You can get the *n*th previous deleted text back in your file by the command

"*n*p

The " here says that a buffer name follows, *n* is the number of the buffer (use the number 1 for now), and **p** is the put command, which puts text in the buffer after the cursor. If this doesn't bring back the text you wanted, type **u** to undo this and then . (period) to repeat the put command. In general the . command repeats the last change you made. When the last command refers to a numbered text buffer, the . command increments the number of the buffer before repeating the command. For example, typing

"1pu.u.u.

will show you all the deleted text which has been saved for you. You can omit the **u** commands here to gather up all this text in the buffer, or stop after any . command to keep just the text recovered so far. The command **P** can be used instead of **p** to put the recovered text before rather than after the cursor.

### 6.4 Recovering Lost Files

If the system crashes, you can recover the work you were doing. You will normally receive mail when you next **login** giving you the name of the file saved for you. You should then change to the directory where you were when the system crashed and give a command of the form:

% **vi** –**r** *name*

replacing *name* with the name of the file you were editing. This recovers your work to a point near where you left off.†

† In some cases, some of the lines of the file may be lost. Vi will give you the numbers of these lines and the text of the lines will be replaced by the string 'LOST'. These lines will almost always be the last few you changed. You can either discard the changes you made (if they are easy to remake) or replace the few lost lines by hand.





You can get a listing of the files saved for you by typing the command:

% **vi** –**r**

If there is more than one copy of a file saved, the visual editor gives you the newest copy each time you recover it. You can get an older saved copy back by first recovering the newer copies.

For this feature to work, **vi** must be correctly installed by a super user on your system, and the **mail** program must receive mail. The command "**vi -r**" will not always list all saved files, but they can be recovered even if they are not listed.

### 6.5 Continuous Text Input

When you are typing in large amounts of text, it is convenient to have lines broken near the right margin automatically. You can set this by giving the command

:se wm=10RETURN

which breaks all lines at least 10 columns from the right edge of the screen.

If **vi** breaks an input line and you wish to put it back together, you can join the lines with **J**. You can give **J** a count of the number of lines to join. For example, **3J** joins 3 lines. **Vi** supplies white space, if appropriate, at the juncture of the joined lines, and leaves the cursor at this white space. You can kill the white space with **x** if you don't want it.

### 6.6 Features for Editing Programs

**Vi** has several commands for editing programs. The thing that distinguishes editing programs from editing text is the need to indent the body of the program. The visual editor's *autoindent* helps you correctly indent programs.

To set this facility, type

:se aiRETURN

Now open a new line with **o**, press the tab key a few times, and type some characters. If you now start another line, the visual editor supplies white space at the beginning of the line to line it up with the previous line. You cannot backspace over this indentation, but you can use **CONTROL-D** to backtab over the indentation.

Each time you type **CONTROL-D**, you back up one position, normally to an 8-column boundary. You can set this with the *shiftwidth* option. Type

:se sw=4RETURN

and then experiment with autoindent again.

To shift lines left and right, use the operators < and >. These shift lines right or left by one *shiftwidth*. Type << and >> which shift one line left or right, and <L and >L which shift the rest of the display left and right.

If you have a complicated expression and want to make sure the parentheses match, put the cursor at a left or right parenthesis and press **%**. This shows you the matching parenthesis. This also works for braces { and }, and brackets [ and ].

If you are editing C programs, you can use the **[[** and **]]** keys to advance or retreat to a line starting with a {, i.e., a function declaration at a time. When you use **]]** with an operator, it stops after a line which starts with }; this is sometimes useful with **y]]**.

### 6.7 Filtering Portions of the Buffer

You can run system commands over portions of the buffer using the operator **!**. You can use this to sort lines in the buffer, or to reformat portions of the buffer with a pretty-printer. Type a list of words, one per line, and end the list with a blank line. Back up to the beginning of the list, and type

!}sortRETURN

which sorts the next paragraph, and the blank line ends a paragraph.





### 6.8 Macros

**Vi** has a parameterless macro facility, which lets you set a single key equal to some longer sequence of keys. You can set this up if you find yourself typing the same sequence of commands repeatedly.

Briefly, there are two kinds of macros:

a) Ones where you put the macro body in a buffer register, for example, *x*. You can then type **@x** to start the macro. You may follow @ with another @ to repeat the last macro.

b) You can use the **map** command from **vi** (typically in your EXINIT) with a command of the form:

:map *lhs rhs*RETURN

mapping *lhs* into *rhs*. There are restrictions: *lhs* should be one keystroke (either one character or one function key) since it must be entered within one second (unless *notimeout* is set, in which case you can type it as slowly as you wish) The *lhs* can be no longer than 10 characters, the *rhs* no longer than 100. To get a space, tab, or newline into *lhs* or *rhs* you should escape them with a **CONTROL-V**. (It may be necessary to double the **CONTROL-V** if the map command is given inside **vi**, rather than in **ex**.) Spaces and tabs inside the *rhs* need not be escaped.

For example, to make **q** write and exit the editor, you can give the command

:map q :wqCONTROL-VCONTROL-VRETURN

which means that whenever you type **q**, it will be as though you had typed the four characters ":wqRETURN". **CONTROL-Vs** are needed because, without them, the RETURN would end the : command, rather than becoming part of the **map** definition. There are two **CONTROL-Vs** because from within **vi**, two **CONTROL-Vs** must be typed to get one. The first RETURN is part of the *rhs*, the second terminates the : command.

You can delete macros with

**unmap** *lhs*

If the *lhs* of a macro is "#0" through "#9", this maps the particular function key instead of the two-character "#" sequence. So that terminals without function keys can access such definitions, the form "#x" means function key *x* on all terminals (and need not be typed within one second.) The character "#" can be changed by using a macro in the usual way. For example, to use tab you could type

:map CONTROL-VCONTROL-VCONTROL-I #

(This won't affect the **map** command, which still uses #, but just the invocation from visual mode).

The undo command reverses an entire macro call as a unit, if it made any changes.

Placing an ! after the word **map** causes the mapping to apply to input mode, rather than command mode. For CONTROL-T to be the same as 4 spaces in input mode, you can type:

:map CONTROL-T CONTROL-Vbbbb

where b is a blank. The CONTROL-V is necessary to prevent the blanks from being taken as white space between the *lhs* and *rhs*.

### 6.9 Word Abbreviations

A feature similar to macros in input mode is word abbreviation. This allows you to type a short word and have it expanded into a longer word or words. The commands are **:abbreviate** and **:unabbreviate** (**:ab** and **:una**) and have the same syntax as **:map**. For example:

:ab eecs Electrical Engineering and Computer Sciences

changes the word "eecs" into the phrase "Electrical Engineering and Computer Sciences". Word abbreviation is different from macros in that only whole words are affected. If "eecs" were typed as part of a larger word, it would be left alone. Also, the partial word is echoed as it is typed. An abbreviation to be any number of keystrokes.

## 7. Nitty-Gritty Details

### 7.1 Line Representation in the Display

Vi folds long logical lines onto many physical lines in the display. Commands which advance lines advance logical lines and will skip over





all the segments of a line in one motion. The command | moves the cursor to a specific column, and may be useful for getting near the middle of a long line to split it in half. Type **80|** on a line which is more than 80 columns long.‡

The visual editor only puts full lines on the display; if there is not enough room on the display to fit a logical line, the visual editor leaves the physical line empty, placing only an @ on the line as a place holder. When you delete lines on a dumb terminal, **vi** often just clears the lines to @ to save time (rather than rewriting the rest of the screen.) You can always maximize the information on the screen by giving the **CONTROL-R** command.

You can have line numbers before each line on the display. Type

:se nuRETURN

to enable this, and

:se nonuRETURN

to turn it off. You can have tabs represented as CONTROL-I and the ends of lines indicated with "$" by giving the command

:se listRETURN

and to turn this off

:se nolistRETURN

Finally, lines consisting of only the character "~" are displayed when the last line in the file is in the middle of the screen. These represent physical lines which are past the logical end of file.

### 7.2 Counts

Most **vi** commands will take a preceding count. The following table gives the common ways counts are used:

‡ You can make long lines very easily by using **J** to join together short lines.

| | |
|---|---|
| new window size | : / ? [[ ]] ` ' |
| scroll amount | CONTROL-D CONTROL-U |
| line/column number | z G \| |
| repeat effect | most of the rest |

Vi maintains the current default window size. On terminals which run at speeds greater than 1200 baud, the visual editor uses the full terminal screen. On terminals which are slower than 1200 baud (most dial-up lines are in this group), the visual editor uses 8 lines as the default window size. At 1200 baud, the default is 16 lines.

This is the size used when vi clears and refills the screen. The commands which take a new window size as count all redraw the screen. If you do not need as large a window as you are currently using, you may change the screen size by specifying the new size before these commands. In any case, the number of lines used on the screen will expand if you move off the top with a − or similar command or off the bottom with a command such as RETURN or CONTROL-D. The window reverts to the last specified size the next time it is cleared and refilled.*

The scroll commands CONTROL-D and CONTROL-U likewise remember the amount of scroll last specified, using half the basic window size initially. The simple insert commands use a count to specify a repetition of the inserted text. Thus

10a+----ESC

inserts a grid-like string of text. A few commands also use a preceding count as a line or column number.

Except for a few commands which ignore any counts (such as CONTROL-R), the rest of the visual editor commands use a count to indicate repetition. Thus "5w" advances five words on the current line, while "5RETURN" advances five lines. A useful example of a count as a repetition is a count given to the . command, which repeats

* But not by a CONTROL-L which just redraws the screen as it is.





the last changing command. If you do **dw** and then "3.", you will delete first one and then three words. Typing "2." deletes two more words.

### 7.3 More File-Manipulation Commands

The following table lists the file-manipulation commands in vi.

| | |
|---|---|
| **:w** | write back changes |
| **:wq** | write and quit |
| **:x** | write (if necessary) and quit (same as **ZZ**). |
| **:e** *name* | edit file *name* |
| **:e!** | reedit, discarding changes |
| **:e +** *name* | edit, starting at end |
| **:e +***n* | edit, starting at line *n* |
| **:e #** | edit alternate file |
| **:w** *name* | write file *name* |
| **:w!** *name* | overwrite file *name* |
| **:***x*,*y***w** *name* | write lines *x* through *y* to *name* |
| **:r** *name* | read file *name* into buffer |
| **:r !***cmd* | read output of *cmd* into buffer |
| **:n** | edit next file in argument list |
| **:n!** | edit next file, discarding changes to current |
| **:n** *args* | specify new argument list |
| **:ta** *tag* | edit file containing tag *tag*, at *tag* |

All of these commands are followed by a RETURN or ESC. The most basic commands are :w and :e. A normal editing session on a single file will end with a **ZZ** command. If you are editing for a long time, you can give :w commands after major edits, and finish with a **ZZ**. When you edit more than one file, you can finish with :w and start editing a new file by typing :e, or set *autowrite* and use **:n** <file>.

If you make changes to vi's copy of a file, but do not want to write them back, then you must type ! after the command you would otherwise use; this forces vi to discard any changes you have made. Use this carefully.

You can add + to the :e command to start at the end of the file, or +*n* to start at line *n*. The *n* may be any command not containing a space, for example, a scan like +/*pat* or +?*pat*. In adding new names to the e command, you can use % which is replaced by the current file name,

or # which is replaced by the alternate file name. The alternate file name is generally the last name you typed other than the current file. If you type :e and get a diagnostic that you haven't written the file, you can give a :w command and then a :e # command to redo the previous :e.

You can write part of the buffer to a file by finding out the lines that bound the range to be written (using CONTROL-G), and giving these numbers after the : and before the **w**, separated by ,'s. For example, "**5,20w fred**" writes lines 5 through 20 to the file "fred". You can also mark these lines with **m** and then use an address of the form *'x,'y* on the **w** command here.

You can read another file into the buffer after the current line by using the **:r** command. You can read in the output from a command, just use **!***cmd* instead of a file name.

To edit a set of files in succession, give all the names on the command line, and then edit each one in turn using the command **:n**. You can also respecify the list of files to edit by giving the **:n** command a list of file names, or a pattern to be expanded as you would have given it on the initial **vi** command.

If you are editing large programs, you will find the **:ta** command very useful. It utilizes a data base of function names and their locations, which can be created by programs, such as **ctags**, to quickly find a function whose name you give. If the **:ta** command requires the editor to switch files, you must **:w** or abandon any changes before switching. You can repeat the **:ta** command without any arguments to look for the same tag again.

### 7.4 More About Searching for Strings

When you are searching for strings in the file with / and ?, **vi** normally places you at the next or previous occurrence of the string. If you are using an operator such as **d**, **c** or **y**, you may want to affect lines up to the line before the line containing the pattern. You can give a search of the form

*/pat/ – n*

to refer to the *n*th line before the next line containing *pat*, or you can





use + instead of – to refer to the lines after the one containing *pat*. If you don't give a line offset, the visual editor affects characters up to the match place, rather than whole lines; use "+0" to affect to the line which matches.

You can have the visual editor ignore the case of words in the searches it does by giving the command

:se icRETURN

The command

:se noicRETURN

turns this off.

Strings given to searches may actually be regular expressions. If you do not want or need this facility, you should

set nomagic

in your EXINIT. When you do this, only the characters ^ and **$** are special in patterns. The character \ is also special (as it is most everywhere in the system), and may be used to get at the an extended pattern matching facility. You must also use \ before a / in a forward scan or a ? in a backward scan. The following table gives the extended forms when *magic* is set.

| | |
|---|---|
| ^ | at beginning of pattern, matches beginning of line |
| $ | at end of pattern, matches end of line |
| . | matches any character |
| \< | matches the beginning of a word |
| \> | matches the end of a word |
| [*str*] | matches any single character in *str* |
| [^*str*] | matches any single character not in *str* |
| [*x*–*y*] | matches any character between *x* and *y* |
| * | matches any number of the preceding pattern |

If you use *nomagic* mode, then the . [ and * primitives are given with a preceding \.

### 7.5 More About Input Mode

There are several characters you use to make corrections during input mode. These are summarized in the following table.

| | |
|---|---|
| CONTROL-H | deletes the last input character |
| CONTROL-W | deletes the last input word, defined as by b |
| erase | your erase character, same as CONTROL-H |
| kill | your kill character, deletes the input on this line |
| \ | escapes a following CONTROL-H and your erase and kill |
| ESC | ends an insertion |
| DEL | interrupts an insertion, terminating it abnormally |
| RETURN | starts a new line |
| CONTROL-D | backtabs over *autoindent* |
| 0CONTROL-D | kills all the *autoindent* |
| ^CONTROL-D | same as 0CONTROL-D, but restores indent next line |
| CONTROL-V | quotes the next non-printing character into the file |

To correct your insertion, type CONTROL-H to correct a single character, or type one or more CONTROL-Ws to back over incorrect words. If you use # as your erase character in the normal system, it will work like CONTROL-H.

Your system kill character, normally @, CONTROL-X, or CONTROL-U, erases all the input you have given on the current line. In general, you can neither erase past a line boundary nor erase characters you did not insert with this insertion command. To make corrections on the previous line, press ESC to end the insertion, make the correction, and then return to where you were. A, which appends at the end of the current line, is often useful for continuing.

To type in your erase or kill character (# or @), precede it with a \, just as you would do at the normal system command level. A more general way of typing non-printing characters into the file is to precede them with a CONTROL-V. The CONTROL-V echoes as a ^ character on which the cursor rests. This indicates that the editor expects you to type a control character. In fact, you may type any character and it will be inserted into the file at that point.†

† This is not quite true. Vi does not allow the NULL (CONTROL-@) character to appear in files. Also the LINEFEED (linefeed or CONTROL-J) character is used by vi to separate lines in the file, so it cannot appear in the middle of a line. You can insert any other character, however, if you wait for vi to echo the ^ before you type the character. In fact, the visual editor will treat a following letter as a request for the corresponding control character. This is the only way to type CONTROL-S or CONTROL-Q, since the system normally uses them to suspend and resume output and never gives them to vi to process.





If you are using *autoindent*, you can backtab over the indent which it supplies by typing CONTROL-D. This backs up to a *shiftwidth* boundary. This only works immediately after the supplied *autoindent*.

When you are using *autoindent*, you may want to place a label at the left margin of a line. The way to do this easily is to type ^ and then CONTROL-D. The editor moves the cursor to the left margin for one line, and restores the previous indent on the next. You can also type a 0 followed immediately by a CONTROL-D to kill all the indent and not have it come back on the next line.

### 7.6 Vi and Ex

Vi is one mode of editing within the editor ex. When you are running vi, you can escape to ex by typing Q. All of the : commands introduced above are available in ex. Likewise, most ex commands can be run from vi using : (colon). Since ex supplies the colon, start the command by typing just the letter and pressing RETURN.

Occasionally, an internal error occurs in vi. If this happens, you get a diagnostic and are left in the command mode of ex. You can then save your work and quit by typing x after the : which ex prompts you with, or you can re-enter vi by giving ex a vi command.

There are several things you can do more easily in ex than in vi. Systematic changes in line-oriented material are particularly easy. You can read the advanced editing documents for the editor ed to find out more about this style of editing. Experienced users often mix their use of ex and vi to speed the work they are doing.

### 7.7 Open Mode: Vi on Hardcopy Terminals and Glass ttys

If you are on a hardcopy terminal or a terminal which does not have a cursor which can move off the bottom line, you can still use the vi command set, but in a different mode. When you give a vi command, the editor will tell you that it is using *open* mode. This name comes from the *open* command in ex, which is used to get into the same mode.

The only difference between *visual* mode and *open* mode is the way in which the text is displayed.

In *open* mode, the editor uses a single line window into the file, and moving backward and forward in the file displays new lines, always below the current line. Two **vi** commands work differently in *open*: **z** and **CONTROL-R**. The **z** command does not take parameters, but rather draws a window of context around the current line and then returns you to the current line.

If you are on a hardcopy terminal, the **CONTROL-R** command retypes the current line. On such terminals, the visual editor normally uses two lines to represent the current line. The first line is a copy of the line as you started to edit it, and you work on the line below this line. When you delete characters, the editor types \'s to show you the characters which are deleted. The visual editor also reprints the current line soon after such changes so that you can see what the line looks like again.

It is sometimes useful to use this mode on very slow terminals which can support **vi** in the full screen mode. You can do this by entering **ex** and typing "open".

# Chapter 3: TEXT EDITOR – ED

CONTENTS

# Chapter 3

# TEXT EDITOR — ED

## 1. Introduction

This is a tutorial to help beginners get started with text editing. UniPlus+ has three text editors: **ed**, **ex**, and **vi**. Of these, **ed** is considered the easiest to learn; however, many users prefer **vi**. We recommend reading this chapter first before going on to **vi**. The **ex** program is the root of the text editors and is used mostly by systems programmers and persons very familiar with **ed**.

Although this chapter does not cover everything about the text editor **ed**, it does discuss enough for most day-to-day needs. This includes:

- Printing, appending, changing, deleting, moving, and inserting text.
- Reading from and writing to files.
- Searching for text.
- Making substitutions.
- Making changes throughout a file automatically.
- Using some special characters for easier editing.

## 2. General

The **ed** program is a text editor—an interactive program for creating and modifying text. The text can be a document (like this one), data for a program, instructions for the computer, etc.

Do the exercises. They illustrate techniques discussed in the text. A summary of commands appears at the end of this chapter.

### 2.1 Disclaimer

We cover only a few **ed** commands. (although we include the most common commands). Also, we assume that you know how to log onto UniPlus+ and understand what a file is. For more information about UniPlus+, refer to the Chapters 1 and 2 in this guide.

You also need to know how to end a line on your terminal. On most terminals, you will end a line by pressing RETURN or the newline key.

remaining words are passed as *arguments* to the command. For example,

```
who
```

is a command that prints the names of users logged in. The command

```
ls -l
```

prints a list of files in the current directory. The argument -l tells ls to print status information, size, and the creation date for each file.

### 2.1 Background Commands

To execute a command, the shell normally creates a new process and waits for it to finish. A command may be run without waiting for it to finish. For example,

```
cc pgm.c &
```

calls the C compiler to compile the file "pgm.c." The trailing "&" is an operator that instructs the shell not to wait for the command to finish. To help keep track of such a process, the shell reports a process number following its creation. A list of currently active processes may be obtained using the **ps** command.

### 2.2 Input/Output Redirection

Most commands produce output to the *standard output* that is initially connected to the terminal. This output may be directed to a file by the notation ">" :

```
ls -l > file
```

The notation "> **file**" is interpreted by the shell and is not passed as an argument to ls. If "file" does not exist, the shell creates it; otherwise, the original contents of "file" are replaced with the output from ls. Output may be appended to the end of a file using the notation ">>" as follows:

```
ls -l >> file
```

In this case, if "file" does not already exist, you will receive an error message.





The *standard input* of a command may be taken from a file instead of the terminal by the notation "<":

```
wc < file
```

The command **wc** reads its standard input (in this case redirected from "file") and prints the number of characters, words, and lines found. If only the number of lines is required, then

```
wc -l < file
```

can be used.

### 2.3 Pipelines and Filters

The standard output of one command may be connected to the standard input of another by writing the *pipe* operator, indicated by |, between commands as in

```
ls -l | wc
```

Two or more commands connected in this way constitute a *pipeline*, and the overall effect is the same as

```
ls -l > file; wc < file
```

except that a file is not used. Instead the two processes are connected by a pipe [see **pipe**(2)] and are run in parallel. Pipes are unidirectional, and synchronization is achieved by halting **wc** when there is nothing to read and halting **ls** when the pipe is full.

A *filter* is a command that reads its standard input, transforms it in some way, and prints the result as output. One such filter, **grep**(1), selects from its input those lines that contain some specified string. For example,

```
ls | grep old
```

prints those lines, if any, of the output from ls that contain the string "old ." Another useful filter is **sort**(1). For example,

```
who | sort
```

will print an alphabetically sorted list of logged in users.

A pipeline may consist of more than two commands, for example,

```
ls | grep old | wc -l
```

prints only the number of file names in the current directory containing the string "old."

### 2.4 File Name Generation

Many commands accept arguments which are file names. For example,

```
ls -l main.c
```

prints only information relating to the file "main.c." The ls -l command alone prints the same information about all files in the current directory.

The shell provides a mechanism for generating a list of file names that match a pattern. For example,

```
ls -l *.c
```

generates as arguments to ls all file names in the current directory that end in ".c." The character "*" is a pattern that will match any string including the null string. In general, *patterns* are specified as follows:

- `*` Matches any string of characters including the null string.
- `?` Matches any single character.
- `[...]` Matches any one of the characters enclosed. A pair of characters separated by a minus will match any character lexically between the pair.

For example,

```
[a-z]*
```

matches all names in the current directory beginning with one of the letters "a" through "z."

The input

```
ls /usr/fred/test/?
```

lists all names in the directory "/usr/fred/test" that consist of a single character. If a file name is not found that matches the pattern, then you will receive the error message "No match."





This mechanism is useful both to save typing and to select names according to some pattern. It may also be used to find files. For example,

```
echo /usr/fred/*/core
```

finds and prints the names of all "core" files in subdirectories of "/usr/fred." [The **echo**(1) command is a standard UniPlus+ command that prints its arguments, separated by blanks.] This last feature can be expensive, requiring a scan of all subdirectories of "/usr/fred."

There is one exception to the general rules given for patterns. The character "." at the start of a file name must be explicitly matched. The input

```
echo *
```

will therefore echo all file names in the current directory not beginning with a period (.). The input

```
echo .*
```

will echo all those file names that begin with "." This avoids inadvertent matching of the names "." and ".." which mean "the current directory" and "the parent directory," respectively. [Notice that ls(1), by default, suppresses information for the files "." and "."]

#### 2.4.1 Quoting

Characters that have a special meaning to the shell, such as

```
< > * ? | &
```

are called *metacharacters*. A complete list of metacharacters is given in Table 5.B. Any character preceded by a \ is *quoted* and loses its special meaning, if any. The \ is omitted so that

```
echo \?
```

will echo a single ?, and

```
echo \\
```

will echo a single \. To allow long strings to be continued over more than one line, the sequence *new line* (or RETURN) is ignored. The \ is convenient for quoting single characters. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between

single quotes. For example,

```
echo xx'****'xx
```

will echo

```
xx****xx
```

The quoted string may not contain a single quote but may contain new lines which are preserved. This quoting mechanism is the most simple and is recommended for casual use. A third quoting mechanism using double quotes is also available and prevents interpretation of some but not all metacharacters.

### 2.5 Prompting by the Shell

When the shell is used from a terminal, it will issue a prompt to the terminal user indicating it is ready to read a command from the terminal. By default, this primary prompt is "**$** ". The prompt may be changed by entering

```
PS1=newprompt
```

This sets the prompt to be the string "newprompt." If a new line is typed and further input is needed, the shell will issue the secondary prompt "> ." Sometimes this can be caused by mistyping a character such as a quote mark. If it is unexpected, then an interrupt (DEL) will return the shell to read another command. This other prompt (">") may be changed by entering:

```
PS2=newprompt2
```

### 2.6 The Shell and Login

Following the user's **login**, the shell is called to read and execute commands typed at the terminal. If the user's **login** directory contains the file ".profile," then it is assumed to contain commands and is read immediately by the shell before reading any commands from the terminal.

### 2.7 Summary

| | |
|---|---|
| **ls** | Prints the names of files in the current directory. |
| **ls > file** | Puts the output from ls into "file." |
| **ls \| wc -l** | Prints the number of files in the current directory. |
| **ls \| grep old** | Prints those file names containing the string "old." |
| **ls \| grep old \| wc -l** | Prints the number of files whose name contains the string "old." |
| **cc pgm.c &** | Runs cc in the background. |

## 3. Shell Procedures

The shell may be used to read and execute commands contained in a file. For example, the following call

**sh** *file* [ *args* ... ]

calls the shell to read commands from "file." Such a file call is called a *command procedure* or *shell procedure*. Arguments may be supplied with the call and are referred to in "file" using the *positional parameters* **$1**, **$2**, ... . For example, if the file "wg" contains

```
who | grep $1
```

then the call

```
sh wg fred
```

is equivalent to

```
who | grep fred
```

All UniPlus+ files have three independent attributes (often called *permissions*), **read**, **write**, and **execute** (rwx). The command **chmod**(1) may be used to make a file executable. For example,

```
chmod +x wg
```

will ensure that the file "wg" has execute status (permission). Following this, the command

```
wg fred
```

is equivalent to the call

```
sh wg fred
```

This allows shell procedures and programs to be used interchangeably. In either case, a new process is created to execute the command.

As well as providing names for the positional parameters, the number of positional parameters in the call is available as **$#**. The name of the file being executed is available as **$0**.

A special shell parameter **$*** is used to substitute for all positional parameters except **$0**. A typical use of this is to provide some default arguments, as in,

```
nroff –T450 –cm $*
```

which simply prepends some arguments to those already given.

### 3.1 Control Flow—for

A frequent use of shell procedures is to loop through the arguments (**$1, $2, ...**) executing commands once for each argument. An example of such a procedure is *tel* that searches the file "/usr/lib/telnos" that contains lines of the form

```
...
fred mh0123
bert mh0789
...
```

The text of *tel* is

```
for i
do
    grep $i /usr/lib/telnos
done
```

The command

```
tel fred
```

prints those lines in "/usr/lib/telnos" that contain the string "fred."

The command

```
tel fred bert
```





prints those lines containing "fred" followed by those for "bert."

The **for** loop notation is recognized by the shell and has the general form

**for** *name* **in** *w1 w2*
**do**
    *command-list*
**done**

A *command-list* is a sequence of one or more simple commands separated or terminated by a new line or a semicolon. Furthermore, reserved words like **do** and **done** are only recognized following a new line or semicolon. A *name* is a shell variable that is set to the words *w1 w2* ... in turn each time the *command-list* following **do** is executed. If "**in** *w1 w2* ..." is omitted, then the loop is executed once for each positional parameter; that is, "**in $***" is assumed.

Another example of the use of the **for** loop is the **create** command whose text is

```
for i do >$i; done
```

The command

```
create alpha beta
```

ensures that two empty files "alpha" and "beta" exist and are empty. The notation "**> file**" may be used on its own to create or clear the contents of a file. Notice also that a semicolon (or new line) is required before **done**.

### 3.2 Control Flow—case

A multiple way (choice) branch is provided for by the case notation. For example,

```
case $# in
    1) cat >>$1 ;;
    2) cat >>$2 <$1 ;;
    *) echo 'usage: append [ from ] to' ;;
esac
```

is an append command. (Note the use of semicolons to delimit the

cases.) When called with one argument as in

```
append file
```

$# is the string "1," and the standard input is appended (copied) onto the end of "file" using the **cat** command.

```
append file1 file2
```

appends the contents of "file1" onto "file2." If the number of arguments supplied to append is other than 1 or 2, then a message is printed indicating proper usage.

The general form of the **case** command is

```
case word in
    pattern) command-list;;
    ...
esac
```

The shell attempts to match *word* with each *pattern* in the order in which the patterns appear. If a match is found, the associated *command-list* is executed and execution of the **case** is complete. Since * is the pattern that matches any string, it can be used for the default case.

**Caution:** No check is made to ensure that only one pattern matches the case argument.

The first match found defines the set of commands to be executed. In the example below, the commands following the second "*" will never be executed since the first "*" executes everything it receives.

```
case $# in
    *) ... ;;
    *) ... ;;
esac
```

Another example of the use of the case construction is to distinguish between different forms of an argument. The following example is a fragment of a **cc** command.





```
for i
do
    case $i in
        -[ocs]) ... ;;
        -*) echo 'unknown flag $i' ;;
        *.c) /lib/c0 $i ... ;;
        *) echo 'unexpected argument $i' ;;
    esac
done
```

To allow the same commands to be associated with more than one pattern, the **case** command provides for alternative patterns separated by a | . For example,

```
case $i in
    -x|-y)...
esac
```

is equivalent to

```
case $i in
    -[xy])...
esac
```

The usual quoting conventions apply so that

```
case $i in
    \?)...
```

will match the character ?.

### 3.2.1 Here Documents

The shell procedure *tel* uses the file "/usr/lib/telnos" to supply the data for **grep**(1). An alternative is to include this data within the shell procedure as a *here* document, as in:

```
for i
do
        grep $i <<!
        ...
        fred mh0123
        bert mh0789
        ...
!
done
```

In this example, the shell takes the lines between <<! and ! as the standard input for **grep**. The string "!" is arbitrary. The document is being terminated by a line that consists of the string following << .

Parameters are substituted in the document before it is made available to **grep** as illustrated by the following procedure called *edg*.

```
ed $3 <<%
g/$1/s//$2/g
w
%
```

The call

```
edg string1 string2 file
```

is then equivalent to the command

```
ed file <<%
g/string1/s//string2/g
w
%
```

and changes all occurrences of "string1" in "file" to "string2." Substitution can be prevented using \ to quote the special character $ as in:

```
ed $3 << +
1,\$s/$1/$2/g
w
+
```

(This version of *edg* is equivalent to the first except that **ed** will print a ? if there are no occurrences of the string **$1**.)





Substitution within a *here* document may be prevented entirely by quoting the terminating string, for example,

```
grep $i <<#
...
#
```

The document is presented without modification to **grep**. If parameter substitution is not required in a *here* document, this latter form is more efficient.

### 3.2.2 Shell Variables

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits, and underscores. Variables may be given values by writing

```
user=fred box=m000 acct=mh0000
```

which assigns values to the variables "user," "box," and "acct." A variable may be set to the null string by entering

```
null=
```

The value of a variable is substituted by preceding its name with $: for example,

```
echo $user
```

will echo "fred."

Variables may be used interactively to provide abbreviations for frequently used strings.

For example,

```
b=/usr/fred/bin
mv file $b
```

will move the "file" from the current directory to the directory "/usr/fred/bin." A more general notation is available for parameter (or variable) substitution, as in:

```
echo ${user}
```

which is equivalent to

```
echo $user
```

and is used when the parameter name is followed by a letter or digit. For example,

```
tmp=/tmp/ps
ps a >${tmp}a
```

will direct the output of ps to the file "/tmp/psa," whereas,

```
ps a >$tmpa
```

would cause the value of the variable "tmpa" to be substituted.

Except for **$?**, the following are set initially by the shell.

**$?** The exit status (return code) of the last command executed as a decimal string. Most commands return a zero exit status if they complete successfully; otherwise, a nonzero exit status is returned. Testing the value of return codes is dealt with later under **if** and **while** commands.

**$#** The number of positional parameters in decimal. Used, for example, in the **append** command to check the number of parameters.

**$$** The process number of this shell in decimal. Since process numbers are unique among all existing processes, this string is frequently used to generate unique temporary file names. For example,

```
ps a >/tmp/ps$$
...
rm /tmp/ps$$
```

**$!** The process number of the last process run in the background (in decimal).

**$–** The current shell flags, such as **–x** and **–v**.

Some variables have a special meaning to the shell and should be avoided for general use.

**$MAIL** When used interactively, the shell looks at the file specified by this variable before it issues a prompt. If the specified file has been modified since it was last looked at,





the shell prints the message "You have mail" before prompting for the next command. This variable is typically set in the file ".profile" in the user's **login** directory. For example:

```
MAIL=/usr/mail/fred
```

**$HOME** The default argument for the **cd** command. The current directory is used to resolve file name references that do not begin with a / and is changed using the **cd** command.

For example,

```
cd /usr/fred/bin
```

makes the current directory "/usr/fred/bin." Then

```
cat wn
```

will print on the terminal the file "wn" in this directory. The command **cd** with no argument is equivalent to

```
cd $HOME
```

This variable is also typically set in the user's **login** profile.

**$PATH** A list of directories containing commands (the *search path*). Each time a command is executed by the shell, a list of directories is searched for an executable file. If **$PATH** is not set, the current directory, "/bin," and "/usr/bin" are searched by default. Otherwise, **$PATH** consists of directory names separated by :. For example,

```
PATH=:/usr/fred/bin:/bin:/usr/bin
```

specifies that the current directory (the null string before the first :), "/usr/fred/bin," "/bin," and "/usr/bin" are to be searched in that order. In this way, individual users can have their own *private* commands that are accessible independently of the current directory. If the command name contains a /, this directory search is not used; a single attempt is made to execute the command.

**$PS1** The primary shell prompt string, by default, "**$** ."

**$PS2** The shell prompt when further input is needed, by default, "> ."

**$IFS** The set of characters used by *blank interpretation*.

### 3.2.3 Test Command

The **test** command is intended for use by shell programs. For example,

```
test –f file
```

returns zero exit status if "file" exists and nonzero exit status otherwise. In general, **test** evaluates a predicate and returns the result as its exit status. Some of the more frequently used **test** arguments are given below [see **test**(1) for a complete specification].

| | |
|---|---|
| test s | true if the argument s is not the null string |
| test –f file | true if "file" exists |
| test –r file | true if "file" is readable |
| test –w file | true if "file" is writable |
| test –d file | true if "file" is a directory |

## 3.3 Control Flow—while

The actions of the **for** loop and the **case** branch are determined by data available to the shell. A **while** or **until** loop and an **if then else** branch are also provided, whose actions are determined by the exit status returned by commands.

A **while** loop has the general form

```
while command-list1
do
    command-list2
done
```

The value tested by the **while** command is the exit status of the last simple command following **while**. Each time round the loop *command-list1* is executed; if a zero exit status is returned, then *command-list2* is executed; otherwise, the loop terminates. For example,





```
while test $1
do
    ...
    shift
done
```

is equivalent to

```
for i
do
    ...
done
```

The **shift** command is a shell command that renames the positional parameters **$2**, **$3**, ... as **$1**, **$2**, ... and loses **$1**.

Another kind of use for the **while/until** loop is to wait until some external event occurs and then run some commands. In an **until** loop, the termination condition is reversed. For example,

```
until test –f file
do
    sleep 300
done
commands
```

will loop until "file" exists. Each time round the loop, it waits for 5 minutes (300 seconds) before trying again. (Presumably, another process will eventually create the file.)

## 3.4 Control Flow—if

Also available is a general conditional branch of the form,

```
if command-list
then
  command-list
else
  command-list
fi
```

that tests the value returned by the last simple command following **if**.

The **if** command may be used in conjunction with the **test** command to test for the existence of a file as in:

```
if test -f file
then
        process file
else
        do something else
fi
```

A multiple test **if** command of the form

```
if ...
then
        ...
else
        if ...
        then
        ...
        else
        if ...
        ...
        fi
        fi
fi
```

may be written using an extension of the **if** notation as,

```
if ...
then
        ...
elif ...
then
        ...
elif ...
...
fi
```

The **touch** command changes the *last modified* time for a list of files. The command may be used in conjunction with **make**(1) to force recompilation of a list of files.





The following example is the **touch** command:

```
flag=
for i
do
        case $i in
                -c)  flag=N ;;
                *)   if test -f $i
                     then
                          ln $i junk$$
                          rm junk$$
                     elif test $flag
                     then
                          echo file \'$i\' does not exist
                     else
                          >$i
                     fi ;;
        esac
done
```

The **-c** flag is used in this command to force subsequent files to be created if they do not already exist. Otherwise, if the file does not exist, an error message is printed. The shell variable *flag* is set to some non-null string if the **-c** argument is encountered. The commands

```
ln ...; rm ...
```

make a link to the file and then remove it.

The sequence

```
if command1
then
        command2
fi
```

may be written

```
command1 && command2
```

Conversely,

```
command1 || command2
```

executes "command2" only if "command1" fails. In each case, the

value returned is that of the last simple command executed.

### 3.5 Command Grouping

Commands may be grouped in two ways,

{ *command-list* ; }

and

( *command-list* )

The first form, *command-list*, is simply executed. The second form executes *command-list* as a separate process. For example,

(cd x; rm junk )

executes "**rm junk**" in the directory "x" without changing the current directory of the invoking shell.

The commands

cd x; rm junk

have the same effect but leave the invoking shell in the directory "x."

### 3.6 Debugging Shell Procedures

The shell provides two tracing mechanisms to help when debugging shell procedures. The first is invoked within the procedure as

set –v

(v for verbose) and causes lines of the procedure to be printed as they are read. It is useful to help isolate syntax errors. It may be invoked without modifying the procedure by entering

sh –v proc ...

where *proc* is the name of the shell procedure. This flag may be used in conjunction with the –**n** flag which prevents execution of subsequent commands. (Note that typing "set –**n**" at a terminal will render the terminal useless until an end-of-file is typed.)

The command

set –x





will produce an execution trace with flag –**x**. Following parameter substitution, each command is printed as it is executed. (Try the above at the terminal to see the effect it has.) Both flags may be turned off by typing

set –

and the current setting of the shell flags is available as **$–**.

### 3.7 The man Command

The following discussion of the **man** command assumes the existence of the document preparation features available as an option on the UniPlus$^+$ system.

The following is the **man** command which is used to print sections of the *UniPlus$^+$ System V User's Manual*. It is called by entering

man sh
man –t ed
man 2 fork

In the first call, the manual section for **sh** is printed. Since no section is specified, section 1 is used. The second call will typeset (–t option) the manual section for **ed**. The last call prints the **fork** manual page from section 2 of the manual.

## 4. Keyword Parameters

Shell variables may be given values by assignment or when a shell procedure is invoked. An argument to a shell procedure of the form *name*=*value* that precedes the command name causes *value* to be assigned to *name* before execution of the procedure begins. The value of *name* in the invoking shell is not affected. For example,

user=fred command

will execute **command** with "user" set to "fred." The –k flag causes arguments of the form *name*=*value* to be interpreted in this way anywhere in the argument list. Such *names* are sometimes called keyword parameters. If any arguments remain, they are available as positional parameters **$1**, **$2**, ... .

The set command may also be used to set positional parameters from within a procedure.

For example,

```
set – *
```

will set $1 to the first file name in the current directory, $2 to the next, etc. Note that the first argument, –, ensures correct treatment when the first file name begins with a –.

### 4.1 Parameter Transmission

When a shell procedure is invoked, both positional and keyword parameters may be supplied with the call. Keyword parameters are also made available implicitly to a shell procedure by specifying in advance that such parameters are to be exported. For example,

```
export user box
```

marks the variables "user" and "box" for export. When a shell procedure is invoked, copies are made of all exportable variables for use within the invoked procedure. Modification of such variables within the procedure does not affect the values in the invoking shell. It is generally true of a shell procedure that it may not modify the state of its caller without an explicit request on the part of the caller. (Shared file descriptors are an exception to this rule.)

Names whose value is intended to remain constant may be declared *readonly*. The form of this command is the same as that of the **export** command,

```
readonly name ...
```

Subsequent attempts to set readonly variables are illegal.

### 4.2 Parameter Substitution

If a shell parameter is not set, then the null string is substituted for it. For example, if the variable "d" is not set,

```
echo $d
```

or





```
echo ${d}
```

will echo nothing. A default string may be given as in:

```
echo ${d–.}
```

which will echo the value of the variable "d" if it is set and "." otherwise. The default string is evaluated using the usual quoting conventions so that

```
echo ${d– '*'}
```

will echo * if the variable "d" is not set. Similarly,

```
echo ${d–$1}
```

will echo the value of "d" if it is set and the value (if any) of $1 otherwise. A variable may be assigned a default value using the notation

```
echo ${d=.}
```

which substitutes the same string as

```
echo ${d–.}
```

and if "d" were not previously set, it will be set to the string ".". (The notation ${... = ...} is not available for positional parameters.)

If there is no sensible default, the notation

```
echo ${d?message}
```

will echo the value of the variable "d" if it has one; otherwise, *message* is printed by the shell and execution of the shell procedure is abandoned. If *message* is absent, a standard message is printed. A shell procedure that requires some parameters to be set might start as follows:

```
: ${user?} ${acct?} ${bin?}
...
```

Colon (:) is a command built into the shell and does nothing once its arguments have been evaluated. If any of the variables "user," "acct," or "bin" are not set, the shell will abandon execution of the procedure.

### 4.3 Command Substitution

The standard output from a command can be substituted in a similar way to parameters. The command **pwd** prints on its standard output the name of the current directory. For example, if the current directory is "/usr/fred/bin," the command

```
d=`pwd`
```

is equivalent to

```
d=/usr/fred/bin
```

The entire string between single quotes (`...`) is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a ` must be escaped using a \.

For example,

```
ls `echo "$1"`
```

is equivalent to

```
ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs (including *here* documents), and the treatment of the resulting text is the same in both cases. This mechanism allows string processing commands to be used within shell procedures. An example of such a command is **basename**, which removes a specified suffix from a string. For example,

```
basename main.c .c
```

will print the string "main." Its use is illustrated by the following fragment from a cc command.

```
case $A in
    ...
    *.c)  B=`basename $A .c`
    ...
esac
```

that sets "B" to the part of "$A" with the suffix ".c" stripped.





Here are some composite examples.

- for i in `ls -t`; do ...

  The variable "i" is set
  to the names of files in time order,
  most recent first.

- set `date`; echo $6 $2 $3, $4

  will print, e.g., "1977 Nov 1, 23:59:59"

### 4.4 Evaluation and Quoting

The shell is a macro processor that provides parameter substitution, command substitution, and file name generation for the arguments to commands. This section discusses the order in which these evaluations occur and the effects of the various quoting mechanisms.

Commands are parsed initially according to the grammar given in Table 5.A. Before a command is executed, the following substitutions occur:

1. Parameter substitution, e.g., **$user**
2. Command substitution, e.g., **`pwd`**

   Only one evaluation occurs so that if, for example, the value of the variable "X" is the string "$y" then

   ```
   echo $X
   ```

   will echo "$y."

3. Blank interpretation

   Following the above substitutions, the resulting characters are broken into nonblank words (*blank interpretation*). For this purpose, *blanks* are the characters of the string "$IFS." By default, this string consists of blank, tab, and newline. The null string is not regarded as a word unless it is quoted. For example,

   ```
   echo ''
   ```

   will pass on the null string as the first argument to **echo**, whereas

   ```
   echo $null
   ```

   will call **echo** with no arguments if the variable "null" is not set or set to the null string.

4. File name generation

   Each word is then scanned for the file pattern characters *, ?, and [...]; and an alphabetical list of file names is generated to replace the word. Each such file name is a separate argument.

The evaluations just described also occur in the list of words associated with a **for** loop. Only substitution occurs in the *word* used for a **case** branch.

As well as the quoting mechanisms described earlier using \ and '...', a third quoting mechanism is provided using double quotes. Within double quotes, parameter and command substitution occurs; but file name generation and the interpretation of blanks does not.

The following characters have a special meaning within double quotes and may be quoted using \.

| | |
|---|---|
| $ | parameter substitution |
| ` | command substitution |
| " | ends the quoted string |
| \ | quotes the special characters $ ` " \ |

For example,

```
echo "$x"
```

will pass the value of the variable "x" as a single argument to **echo**. Similarly,

```
echo "$*"
```

will pass the positional parameters as a single argument and is equivalent to

```
echo "$1 $2 ..."
```

The notation $@ is the same as $* except when it is quoted. Inputting

```
echo "$@"
```

will pass the positional parameters, unevaluated, to **echo** and is equivalent to





```
echo "$1" "$2" ...
```

The following illustration gives, for each quoting mechanism, the shell metacharacters that are evaluated.

| quoting mechanism | metacharacters \ | $ | * | ` | " | ' |
|---|---|---|---|---|---|---|
| ' | n | n | n | n | n | t |
| ` | y | n | n | t | n | n |
| " | y | y | n | y | t | n |

t = terminator
y = interpreted
n = not interpreted

In cases where more than one evaluation of a string is required, the built-in command **eval** may be used. For example, if the variable "X" has the value "$y" and if "y" has the value "pqr," then

```
eval echo $X
```

will echo the string "pqr."

In general, the **eval** command evaluates its arguments (as do all commands) and treats the result as input to the shell. The input is read and the resulting command(s) executed. For example,

```
wg='eval who | grep'
$wg fred
```

is equivalent to

```
who | grep fred
```

In this example, **eval** is required since there is no interpretation of metacharacters, such as |, following substitution.

### 4.5 Error Handling

The treatment of errors detected by the shell depends on the type of error and on whether the shell is being used interactively. An interactive shell is one whose input and output are connected to a terminal [as

determined by **gtty**(2)]. A shell invoked with the **–i** flag is also interactive.

Execution of a command may fail for any of the following reasons:

- Input/output redirection may fail. For example, if a file does not exist or cannot be created.
- The command itself does not exist or cannot be executed.
- The command terminates abnormally, for example, with a *bus error* or *memory fault* signal.
- The command terminates normally but returns a nonzero exit status.

In all of these cases, the shell will go on to execute the next command. Except for the last case, an error message will be printed by the shell. All remaining errors cause the shell to exit from a command procedure. An interactive shell will return to read another command from the terminal. Such errors include the following:

- Syntax errors, e.g., **if ... then ... done**
- A signal such as interrupt. The shell waits for the current command, if any, to finish execution and then either exits or returns to the terminal.
- Failure of any of the built-in commands such as **cd**(1).

The shell flag **–e** causes the shell to terminate if any error is detected. The following is a list of the UniPlus+ signals:

| | |
|---|---|
| 1 | hangup |
| 2 | interrupt |
| 3* | quit |
| 4* | illegal instruction |
| 5* | trace trap |
| 6* | IOT instruction |
| 7* | EMT instruction |





| | |
|---|---|
| 8* | floating point exception |
| 9 | Kill (cannot be caught or ignored) |
| 10* | bus error |
| 11* | segmentation violation |
| 12* | bad argument to system call |
| 13 | write on a pipe with no one to read it |
| 14 | alarm clock |
| 15 | software termination [from **kill**(1)] |

The UniPlus+ system signals marked with an asterisk "*" as shown in the list produce a core dump if not caught. However, the shell itself ignores quit which is the only external signal that can cause a dump. The signals in this list of potential interest to shell programs are 1, 2, 3, 14, and 15.

### 4.6 Fault Handling

Shell procedures normally terminate when an interrupt is received from the terminal. The **trap** command is used if some cleaning up is required, such as removing temporary files. For example,

```
trap 'rm /tmp/ps$$; exit' 2
```

sets a trap for signal 2 (terminal interrupt); and if this signal is received, it will execute the following commands:

```
rm /tmp/ps$$; exit
```

The **exit** is another built-in command that terminates execution of a shell procedure. The **exit** is required; otherwise, after the trap has been taken, the shell will resume executing the procedure at the place where it was interrupted.

UniPlus+ system signals can be handled in one of three ways.

1. They can be ignored, in which case the signal is never sent to the process.
2. They can be caught, in which case the process must decide what action to take when the signal is received.

3. They can be left to cause termination of the process without it having to take any further action.

If a signal is being ignored on entry to the shell procedure, for example, by invoking it in the background, **trap** commands (and the signal) are ignored.

The use of **trap** is illustrated by this modified version of the **touch** command illustrated below:

```
flag=
trap 'rm -f junk$$; exit' 1 2 3 15
for i
do
    case $i in
    -c) flag=N ;;
    *)  if test -f $i
        then
            ln $i junk$$; rm junk$$
        elif test $flag
        then
            echo file \'$i\' does not exist
        else
            >$i
        fi ;;
    esac
done
```

The cleanup action is to remove the file "junk$$." The **trap** command appears before the creation of the temporary file; otherwise, it would be possible for the process to die without removing the file.

Since there is no signal 0 in +, it is used by the shell to indicate the commands to be executed on exit from the shell procedure.

A procedure may, itself, elect to ignore signals by specifying the null string as the argument to trap. The following:

```
trap '' 1 2 3 15
```

is a fragment taken from the **nohup**(1) command which causes the





system HANGUP, INTERRUPT, QUIT, and SOFTWARE TERMINATION signals to be ignored both by the procedure and by invoked commands.

Traps may be reset by entering

```
trap 2 3
```

which resets the traps for signals 2 and 3 to their default values. A list of the current values of traps may be obtained by writing

```
trap
```

The **scan** procedure is an example of the use of **trap** where there is no exit in the trap command. The **scan** takes each directory in the current directory, prompts with its name, and then executes commands typed at the terminal until an end of file or an interrupt is received. Interrupts are ignored while executing the requested commands but cause termination when **scan** is waiting for input. The **scan** procedure follows:

```
d=`pwd`
for i in *
do
    if test -d $d/$i
    then
        cd $d/$i
        while echo "$i:" && trap exit 2 && read x
        do
            trap : 2
            eval $x
        done
    fi
done
```

The "**read x**" is a built-in command that reads one line from the standard input and places the result in the variable "x." It returns a nonzero exit status if either an end-of-file is read or an interrupt is received.

### 4.7 Command Execution

To run a command (other than a built-in), the shell first creates a new process using the system call **fork**(2). The execution environment for the command includes input, output, and the states of signals and is

established in the child process before the command is executed. The built-in command **exec** is used in rare cases when no fork is required and simply replaces the shell with a new command. For example, a simple version of the **nohup** command looks like

```
trap '' 1 2 3 15
exec $*
```

The **trap** turns off the signals specified so that they are ignored by subsequently created commands, and exec replaces the shell by the command specified.

Most forms of input/output redirection have already been described. In the following, *word* is only subject to parameter and command substitution. No file name generation or blank interpretation takes place so that, for example,

```
echo ... >*.c
```

will write its output into a file whose name is "*.c." Input/output specifications are evaluated left to right as they appear in the command. Some input/output specifications are as follows:

| | |
|---|---|
| > *word* | The standard output (file descriptor 1) is sent to the file "word" which is created if it does not already exist. |
| >> *word* | The standard output is sent to file "word." If the file exists, then output is appended (by seeking to the end); otherwise, the file is created. |
| < *word* | The standard input (file parameter 0) is taken from the file "word." |
| << *word* | The standard input is taken from the lines of shell input that follow up to but not including a line consisting only of "word." If "word" is quoted, no interpretation of the document occurs. If "word" is not quoted, parameter and command substitution occur and \ is used to quote the characters \, $, `, and the first character of "word." In the latter case, **newline** is ignored (e.g., quoted strings). |
| >& *digit* | The file descriptor "digit" is duplicated using the system call **dup**(2), and the result is used as the standard output. |
| <& *digit* | The standard input is duplicated from file descriptor "digit." |
| <&– | The standard input is closed. |
| >&– | The standard output is closed. |





Any of the above may be preceded by a digit in which case the file descriptor created is that specified by the digit instead of the default 0 or 1. For example,

```
... 2>file
```

runs a command with message output (file descriptor 2) directed to "file." Another example,

```
... 2>&1
```

runs a command with its standard output and message output merged. (Strictly speaking, file descriptor 2 is created by duplicating file descriptor 1; but the effect is usually to merge the two streams.)

The environment for a command run in the background such as

```
list *.c | lpr &
```

is modified in two ways. First, the default standard input for such a command is the empty file "/dev/null." This prevents two processes (the shell and the command), which are running in parallel, from trying to read the same input. Chaos would ensue if this were not the case. For example,

```
ed file &
```

would allow both the editor and the shell to read from the same input at the same time.

The other modification to the environment of a background command is to turn off the QUIT and INTERRUPT signals so that they are ignored by the command. This allows these signals to be used at the terminal without causing background commands to terminate. For this reason, the UniPlus$^+$ convention for a signal is that if it is set to 1 (ignored), then it is never changed even for a short time. Note that the shell command **trap** has no effect for an ignored signal.

### 4.8 Invoking the Shell

The following flags are interpreted by the shell when it is invoked. If the first character of argument zero is a minus, commands are read from the file ".profile."

**–c** *string* If the **–c** flag is present, then commands are read from "string."

**–s** If the **–s** flag is present or if no arguments remain, commands are read from the standard input. Shell output is written to file descriptor 2.

**–i** If the **–i** flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case, TERMINATE is ignored (so that "**kill 0**" does not kill an interactive shell, and INTERRUPT is caught and ignored (so that **wait** is interruptible). In all cases, QUIT is ignored by the shell.





## 5. Tables

TABLE 5.A. Grammar

| | |
|---|---|
| *item* | *word*<br>*input-output*<br>*name* = *value* |
| *simple-command:* | *item*<br>*simple-command item* |
| *command:* | *simple-command*<br>**(** *command-list* **)**<br>**{** *command-list* **}**<br>**for** *name* **do** *command-list* **done**<br>**for** *name* **in** *word* ... **do** *command-list* **done**<br>**while** *command-list* **do** *command-list* **done**<br>**until** *command-list* **do** *command-list* **done**<br>**case** *word* **in** *case-part* ... **esac**<br>**if** *command-list* **then** *command-list else-part* **fi** |
| *pipeline:* | *command*<br>*pipeline* **\|** *command* |
| *andor:* | *pipeline*<br>*andor* **&&** *pipeline*<br>*andor* **\|\|** *pipeline* |
| *command-list:* | *andor*<br>*command-list* **;**<br>*command-list* **&**<br>*command-list* **;** *andor*<br>*command-list* **&** *andor* |
| *input-output:* | **>** *word*<br>**<** *word*<br>**>>** *word*<br>**<<** *word* |
| *file* | *word*<br>**&** *digit*<br>**& –** |
| *case-part:* | *pattern* **)** *command-list* **;;** |
| *pattern:* | *word*<br>*pattern* **\|** *word* |

| | |
|---|---|
| *else-part:* | **elif** *command-list* **then** *command-list else-part*<br>**else** *command-list* |
| *empty:* | *empty* |
| *word:* | sequence of nonblank characters |
| *name* | sequence of letters, digits, or underscores starting with a letter |
| *digit:* | **0 1 2 3 4 5 6 7 8 9** |





**TABLE 5.B.** Metacharacters and Reserved Words

(a) *syntactic:*

| | |
|---|---|
| \| | pipe symbol |
| && | 'andf' symbol |
| \|\| | 'orf' symbol |
| ; | command separator |
| ;; | case delimiter |
| & | background commands |
| ( ) | command grouping |
| < | input redirection |
| << | input from a here document |
| > | output creation |
| >> | output append |

(b) *patterns:*

| | |
|---|---|
| * | match any character(s) including none |
| ? | match any single character |
| [...] | match any of the enclosed characters |

(c) *substitution:*

| | |
|---|---|
| ${...} | substitute shell variable |
| `...` | substitute command output |

(d) *quoting:*

| | |
|---|---|
| \ | quote the next character |
| '...' | quote the enclosed characters except for the ' |
| "..." | quote the enclosed characters except for the $, ` ,\, and " |

(e) *reserved words:*

if then else elif fi

case in esac

for while until do done

{ } [ ] test



# Chapter 6: THE C SHELL — CSH

CONTENTS

# Chapter 6

# THE C SHELL — CSH

## 1. Introduction

The C shell (**csh**) is a new command language interpreter for UNIX systems. It incorporates useful features of other shells and a **history** mechanism. While incorporating many features of other shells which make writing shell programs (shell scripts) easier, most of the features unique to **csh** are designed more for the interactive UNIX user.

Users who have read a general introduction to the system will find a valuable basic explanation of the shell here. Simple terminal interaction with **csh** is possible after reading just the first part of this chapter. The second part describes the shell's capabilities which you can explore after you have begun to become acquainted with the shell. Later sections introduce features which are useful, but not necessary for all users of the shell.

There is an appendix listing of special characters of the shell at the end of this chapter. In addition, the **csh**(1) entry in the *UniPlus+ System V User's Manual* provides a description of the features of the C shell and is a final reference for questions about the C shell.

## 2. Using the C Shell

A *shell* is a command language interpreter. **Csh** is the name of one particular command interpreter of UniPlus+. The primary purpose of **csh** is to translate command lines typed at a terminal into system actions, such as invoking other programs. **Csh** is a user program just like any you might write. We hope **csh** will be a very useful program for you in interacting with the UniPlus+ system. While **csh** has a set of built-in functions which it performs directly, most commands cause execution of programs that are, in fact, external to the shell. The shell is thus distinguished from the command interpreters of other systems both by the fact that it is just a user program, and by the fact that it is used almost exclusively as a mechanism for invoking other programs.

## 2.1 Basic Notion of Commands

*Commands* in the UniPlus+ system expect a list of strings or words as *arguments*. Thus the *command line*

```
% mail bill
```

consists of two words and the prompt "%". The first word "mail" is the name of the command to be executed, in this case the **mail** program which sends messages to other users. The shell uses the name of the command in attempting to execute the program for you. It will look in a number of *directories* for a *file* with the name "mail" which is expected to contain the **mail** program.

The rest of the words on the command line are given as arguments to the command. In this case, the argument "bill" is interpreted by the **mail** program as the name of a user to whom the message is to be sent. The following is an example of using the **mail** command (the % is the C shell prompt):

```
% mail bill
Let's have lunch to discuss the changes proposed
at this morning's meeting.
How does tomorrow noon sound?
                maf
EOT
%
```

The prompt (%) indicates that the shell is ready to read input from the terminal. After the command line (*mail bill*) was input, the shell executed the **mail** program with argument "bill" and went dormant waiting for it to complete. The **mail** program then read input from the terminal as the message to be sent until signaled with an end-of-file. (An end-of-file is indicated with CONTROL-d i.e., striking both the CONTROL and "d" keyboard characters simultaneously.) The **mail** program was then completed and the shell issued another prompt (%) to indicate that it was ready to read from the terminal again.

This is the essential pattern of all interaction with the UniPlus+ system through the shell:

1. A complete command is typed at the terminal,
2. the shell executes the command,
3. then the shell prompts for a new command.

If you run the editor for an hour, the shell will patiently wait for you to finish editing and obediently prompt you again whenever you finish editing.

An example of a useful command you can execute now is the **tset** command, which sets the default *erase* and *kill* characters on your terminal—the erase character erases the last character you typed and the kill character erases the entire line you have entered so far. By default, in the shell, the erase character is "#" and the kill character is "@". Most people who use CRT terminals prefer to use the backspace character keyboard character or CONTROL-h as their erase character. To change the erase character to backspace, give the following command:

```
% tset -e
```

which tells the program **tset** to set the erase character (**-e**) to the default setting which is the backspace character [see **tset**(1)]. It should be noted that CONTROL-h and backspace are used interchangeably here due to the fact that, on most terminals, the backspace key sends a CONTROL-h character to the system.

## 2.2 Flag Arguments

A useful notion is that of a *flag* or *option* argument to programs. While many arguments to commands specify file names or user names, some arguments specify an optional capability of the command which you wish to invoke. By convention, such arguments begin with the hyphen character (**-**). For example, the command:

```
% ls
```

will produce a list of the files located in the current *working directory*. The **-s** is the *size* option to **ls**. Thus:

```
% ls -s
```

will produce a list of the file names as well as the size of the file in blocks of 512 characters. The manual entry for each command in the *UniPlus+ System V User's Manual* gives the available options for each command. The **ls** command has a large number of useful and

interesting options.

### 2.3 Output to Files

Commands that normally read input or write output on the terminal can also be executed with this input and/or output done to a file.

Suppose you wish to save the current date in a file called "now". The command

```
% date
```

prints the current date on the terminal. This is because the terminal is the default *standard output*. The shell lets us *redirect* the standard output of a command through a notation using the special character ">" and the name of the file where output is to be placed. Thus the command:

```
% date > now
```

runs the **date** command such that its standard output is the file "now" rather than the terminal. Therefore, it places the current date and time into the file "now". It is important to know that the **date** command is unaware that its output is going to a file rather than to the terminal. The shell performed this redirection before the command began executing.

If the file "now" did not exist before **date** was executed, the shell would have created the file. If the file already existed, the previous version would have been discarded. **Note:** A shell option **noclobber** prevents this from happening accidentally.

The default is for files to be permanent, meaning that it will remain until removed by the user. If you wish to create a file which will be removed automatically, you can begin the file name with a "#" symbol character*, this *scratch* character denotes that the file may be discarded after a few days or even sooner if file space becomes tight. Thus, using the example above, the file "now" probably does not need to be saved

* Note that if your erase character is a #, you will have to precede the # with a \.





forever, so it would be more appropriate to give the following command:

```
% date > #now
```

### 2.4 Input from Files

We discussed above how to redirect the standard output of a command to a file. It is also possible to redirect the *standard input* of a command from a file. This is not often necessary since most commands will read from a file whose name is given as an argument. Using the **mail** program from our earlier example, we could have put the message to "bill" in a file called "note" and sent the message by giving the following command:

```
% mail bill < note
```

The **mail** program would accept as the message the contents of the file "note", rather than going dormant and waiting for the message to be input from the terminal. In this case, a CONTROL-d would not be necessary to terminate **mail**.

It is not often necessary to route the standard input of a command from a file since most commands will read from a file whose name is given as an argument. For example, assuming there exists a file called "data", the command lines:

```
% sort < data
```

or

```
% sort data
```

would sort the contents of "data" alphabetically, line by line. The latter command line lets **sort** open the file "data" itself and, since it is less to type, is preferred.

You should note, however, that the command line:

```
% sort
```

would sort lines from the standard input. Since we did not redirect the standard input, it would sort lines as we typed them on the terminal until we entered a CONTROL-d to indicate an end-of-file, at which point it would print the alphabetized list on the standard output.

## 2.5 Metacharacters in the C Shell

The shell has a large number of special characters (like the redirection characters ">" and "<") which indicate special functions. These notations have syntactic and semantic meaning to the shell and are called *metacharacters*. In general, most characters which are neither letters nor digits have special meaning to the shell. We shall shortly learn a means of *quotation* which allows use of these *metacharacters* without the shell treating them in any special way.

Metacharacters normally have effect only when the shell is reading input. You need not worry about placing shell metacharacters in the text of a letter you are sending via **mail**, or when typing in text or data to some other program. Note that the shell is only reading input when a prompt (%) has been displayed.

### 2.5.1 Pipelines

A useful capability that UniPlus+ has is to combine the standard output of one command with the standard input of another, i.e., to run the commands in a sequence known as a *pipeline*. For instance, the command:

```
% ls -s
```

normally produces a list of the files in a directory with the size of each in blocks of 512 characters. If we are interested in learning which of our files is the largest, we may prefer to have this list sorted by size. The default in which **ls** lists is ASCII. Checking the options of ls, we see that there isn't a specific option to list by number within ls. However, the **sort** command has just the options required. We can combine the **ls** command and the **sort** command with options and get what we want.

The **-n** option of **sort** specifies a numeric sort rather than an alphabetic sort. Thus:

```
% ls -s | sort -n
```

specifies that the output of the ls command run with the option **-s** (size) is to be *piped* to the command **sort** with the **-n** (numeric) option. This would give produce a sorted list of our files by size, but with the smallest size first. We could then use the **-r** (reverse) sort option:





```
% ls -s | sort -n -r
```

Reading this command line left to right, first the contents of the directory were listed, each with the size (**-s**) in blocks. Then this was given as the standard input to the **sort** command asking it to sort numerically (**-n**) in reverse order (**-r**) (largest first).

The notation introduced above is called the *pipe* mechanism. Commands separated by a "|" character are connected together by the shell and the standard output of each is run into the standard input of the next. The leftmost command in a pipeline will normally take its standard input from the terminal and the rightmost will place its standard output on the terminal. Other examples of pipelines will be given later when we discuss the history mechanism.

### 2.5.2 Pathnames

Many commands to be executed will need the names of files as arguments. *Pathnames* consist of a number of components separated by a "/". Each component (except the last, which is the file name) names a directory in which the next component resides; in effect, specifying the *path* of directories to follow to reach the file. Thus the pathname:

```
/etc/motd
```

specifies the file "motd" in the directory "etc" which is a subdirectory of the *root* directory "/". A pathname that begins with a slash is said to be an *absolute pathname* since it is specified from the absolute top of the entire directory hierarchy of the system (the root). Pathnames which do not begin with "/" are interpreted as starting in the current *working directory*, which is initially your *home* directory and can be changed dynamically by the **cd** (change directory) command. Such pathnames are said to be relative to the working directory since they are found by starting in the working directory and descending to lower levels of directories for each component of the pathname. If the pathname contains no slashes at all, then the file is contained in the working directory itself and the pathname is merely the name of the file in this directory. Absolute pathnames, those beginning with "/", have no relation to the working directory.

### 2.5.3 File Names

Most file names consist of a number of alphanumeric characters and periods. In fact, all ASCII characters except "/" (slash) may appear in file names. It is inconvenient to have most non-alphanumeric characters in file names because many of these have a special meaning to the shell.

Note: Files with the character "." at the beginning are treated specially. This prevents accidental matching of the file names "." and ".." in the working directory, which have special meaning to the system, as well as other files such as ".cshrc", which are not normally visible.

The period character is not a shell-metacharacter and is often used to separate the *extension* of a file name from the base of the name. For example:

```
prog.c  prog.o  prog.err  prog.out
```

are four related files. They share a *base* portion of a name which is part of the name that is common to all of the related file and is left when a trailing "." and the extension (the characters following the period) is stripped off. The file "prog.c" might be the source for a C program, the file "prog.o" the corresponding object file, the file "prog.err" the errors resulting from a compilation of the program, and the file "prog.out" the output of a run of the program.

### 2.5.4 Asterisk

Using the above example, if we wished to refer to all four of files in a command, we could use the notation

```
prog.*
```

This notation is expanded by the shell into a list of names which begin with "prog.". The asterisk (*) matches any sequence (including the empty sequence) of characters in a file name. The names which match are placed in the *argument list* of the command. Thus, the command:

```
% echo prog.*
```

will echo (print on the standard output) the names





```
prog.c  prog.err  prog.o  prog.out
```

The **echo** command receives four words as arguments, even though we only typed one word as as argument directly. The four words were generated by the *filename expansion* mechanism of the C shell on the one input word.

### 2.5.5 Question Mark

Other notations for *filename expansion* are also available. The character "?" matches any *single* character in a file name. Thus:

```
% echo ? ?? ???
```

will return a list of file names: first those with one character names, then those with two character names, and finally those with three character names. The file names of each of the lengths will be independently sorted. Also, the command:

```
% echo prog.? prog.?? prog.???
```

will match and echo files which begin with the base "prog." and end in one character, then two characters and finally those that end with three characters. If there were no match for "prog.??", but there are matches for both "prog.?" and "prog.???", the shell would not return a "No match" message. However, it would do so if it found no match to any of the file name requests.

### 2.5.6 Brackets

Another mechanism consists of a sequence of characters between "[" and "]". This *metasequence* matches any single character from the enclosed set. Thus:

```
prog.[co]
```

will match the files:

```
prog.c prog.o
```

in the example above. We can also place two characters around a "-" in this notation to denote a range. For example,

```
chap.[1-5]
```

would match files

chap.1 chap.2 chap.3 chap.4 chap.5

if they existed. This is a shorthand notation for

chap.[12345]

and otherwise equivalent.

As another example of the use of brackets, the notation:

chap.[a-z]

would match "chap." files that had extensions of a single lowercase letter The notation "[A-Z]" could be used for an uppercase match and "[a-z,A-Z]" to match any letter, regardless of case.

If an argument list to a command contains filename expansion syntax and this syntax fails to match any existing file names (and the C shell variable *nonomatch* is not set), then the shell considers this to be an error and prints a diagnostic

No match.

and does not execute the command.

### 2.5.7 Tilde

Another filename expansion mechanism gives access to the pathname of the *home directory* of other users. This notation consists of the tilde character (~) followed by user's **login** name. For instance,

~bill

would map to the home directory of "bill". Since, on large systems, users may have **login** directories scattered over many different disk volumes with different prefix directory names, this notation provides a reliable way of accessing the files of other users.

A special case of this notation consists of using "~" alone. This notation is expanded by the shell into your home directory. This can be very useful if you have used **cd** to change to another directory and have found a file you wish to copy into your home directory. The command:

% cp thatfile ~

would copy "thatfile" into your home directory.





### 2.5.8 Braces

There also exists a mechanism using the characters "{" and "}" for abbreviating a set of words which have common parts. This mechanism is described in the section "Other, Less Commonly Used, C Shell Features."

### 2.5.9 Escaping Metacharacters

We have already seen a number of metacharacters used by the shell. These metacharacters pose a problem as we cannot use them directly as parts of words. Thus, the command:

% echo *

will not echo the character "*". It will either echo a list of file names in the current working directory, or print the message "No match" if there are no files in the working directory.

The method for inhibiting the expansion syntax of metacharacters is to enclose it in single quotation characters, i.e.,

% echo '*'

There is a special character "!" which is used by the **history** mechanism of the shell and which cannot be *escaped* by placing it within single quotes. It and the character "'" itself can be preceded by a single "\" to prevent their special meaning. Thus:

% echo \'\!

prints:

'!

Also,

% echo \''*'

prints:

'*

since the first "\" escaped the first "'" and the "*" was enclosed between "'" characters.

## 2.6 Terminating Commands

When you are executing a command and the shell is waiting for it to complete, there are several ways to force it to stop. For instance, if you type the command:

```
% cat /etc/passwd
```

the system will concatenate (print) a copy of a list of all users of the system on your terminal. This is likely to continue for several minutes unless you stop it. You can send an *interrupt* signal to the cat command by typing the DELETE, DEL or RUBOUT key on your terminal.* Since **cat** does not take any precautions to avoid or otherwise handle this signal, the interrupt will cause it to terminate. The shell notices that **cat** has terminated and displays another prompt (%). If you hit an interrupt again, the shell will just repeat its prompt since it handles interrupt signals and chooses to continue to execute commands rather than terminating, which would have the effect of logging you out.

Another way in which many programs terminate is when they get an end-of-file from their standard input. The shell also terminates when it receives an end-of-file. This can happen when you input a CONTROL-d at a prompt. This means that typing CONTROL-d one too many times could cause you to **logout** accidently, however, the shell has a mechanism for preventing this. This option is called **ignoreeof**. You can set this option in your ".cshrc" file and the shell will then effectively "ignore the (eof) end-of-file" sent by a CONTROL-d character. If this option is set, you must give the command **logout** to log off the system.

If a command has its standard input redirected from a file, then it will normally terminate when it reaches the end of this file. Thus, if we execute:

```
mail bill < note
```

the **mail** will terminate without typing a CONTROL-d. This is because it read to the end of the file "note".

* Many users use *stty*(1) to change the interrupt character to CONTROL-c.





If you write or run programs which are not fully debugged, then it may be necessary to stop them somewhat ungracefully. This can be done by sending them a *quit* signal, sent by typing a CONTROL-\. This will usually provoke the shell to produce a message like:

```
Quit (Core dumped)
```

indicating that a file "core" has been created containing information about the program state when it terminated due to the quit signal. You can examine this file yourself, or forward information to the maintainer of the program telling him/her where the *core file* is.

Background commands will ignore interrupt and quit signals from the terminal. To stop these processes you must use the **kill** command. For example,

```
% nroff bigfile &
17594
% kill 17594
```

The process "**nroff bigfile**" was put in the background with the "&" symbol and was followed by the *process number*. Then, the **kill** command was issued. The **kill** command accepts the process number as an argument (without a %). The **ps** command can be used to find out the process number.

If you want to examine the output of a command, but realize that the output will be more than one screenful, you can use the **more** command. For example,

```
% more /etc/passwd
```

will print the contents of "/etc/passwd" on the standard output one screenful at a time. The **more** program pauses after each complete screenful and types "--More--". You then hit the space bar to get another screenful, a RETURN to get another line, or a "q" to end the **more** program. You can also use more as a *filter*, i.e.

```
% sort /etc/passwd | more
```

which pipes the standard output from the **sort** command through the **more** program and prints that output one screenful at a time.

For stopping output of commands not involving **more**, you can use the CONTROL-s key to stop the output. The output will resume when you hit CONTROL-q (or any other key), but CONTROL-q is normally used because it only restarts the output and does not become input to the program which is running. This works well on low-speed terminals, but at 9600 baud it is hard to type CONTROL-s and CONTROL-q fast enough to paginate the output nicely, and a program like **more** is usually used.

## 3. Details of the C Shell

Much of the discussion in the preceding sections has been applicable to both the shell (**sh**) and the C shell (**csh**) of the UniPlus+ operating system. The following sections will introduce many features particular to **csh** so you will need to be using the C shell to try these features. The shell that you are using when you **login** to the system is specified in the "/etc/passwd" file [see **passwd(4)**]. If you are using the shell "/bin/sh", you can switch to the C shell by typing the following command line:

**chsh** *your_name* /bin/csh

You only need to do this once; it takes effect at all subsequent **logins**.

### 3.1 Startup and Termination

When you **login** the C shell is started by the system and begins by reading commands from a file ".cshrc" in your home directory. All C shell commands which you may execute during your terminal session will read from this file. We will later see what kinds of useful commands are placed there. However, the C shell does not complain about its absence, and for now you need not have this file.

After you **login** to the system, the C shell will, after it reads commands from ".cshrc", read commands from a file ".login" also in your home directory. This file contains commands which you wish to do each time you **login** to the UniPlus+ system. A sample ".login" file looks something like:

```
set ignoreeof
set noclobber
set mail=(/usr/spool/mail/bill)
set time=15 history=10
stty 9600
```

This file contains several commands to be executed at the time of logging into the system. The first is a **set** command which is interpreted directly by the C shell. It sets the C shell variables *ignoreeof* (which causes the C shell to not log out if a CONTROL-d is hit), *noclobber* (which prevents overwriting of existing files), and *mail* (which causes the C shell to watch every 5 minutes for incoming mail to tell the user if more mail has arrived). Next the C shell variable *time* is set to "15" (which causes the C shell to automatically print out statistics lines for commands which execute for at least 15 seconds of CPU time), and *history* is set to "10" (which indicates that user wants the C shell to remember the last 10 commands given). The **stty** command is used to specify the terminal baud rate to 9600. When the stty program is finished, the C shell finishes processing the **.login** file and begins reading commands from the terminal, prompting for each with a percent (%).

When logging off by giving the **logout** command, the C shell will print "logout" and execute commands from the file ".logout" if it exists in the home directory. The C shell will then terminate and you will be logged off the system. From this point, the C shell will no longer read commands from the terminal. If the system is not going down, you will receive a new **login** message.

### 3.2 C Shell Variables

The C shell maintains a set of *variables*. We saw above the variables *time* and *history* which had values "15" and "10". In fact, each C shell variable has as value an array of zero or more *strings*. Shell variables may be assigned values by the **set** command. It has several forms, the most useful of which was given above and is

set name=value

The C shell variables may be used to store values which are to be used in commands later through a substitution mechanism. The C shell variables most commonly referenced are, however, those which the C

shell itself refers to. By changing the values of these variables one can directly affect the behavior of the C shell.

One of the most important variables is the variable *path*. This variable contains a sequence of directory names where the C shell searches for commands. The **set** command with no arguments shows the value of all variables currently defined (we usually say set) in the C shell. The default value for *path* will be shown by **set**, for example:

```
% set
argv    ()
home    /usr/bill
path    (. /usr/ucb /bin /usr/bin)
prompt  %
shell   /bin/csh
status  0
%
```

This output indicates that the *path* variable points to the current directory "." and then "/usr/ucb", "/bin" and "/usr/bin". Commands which you may write might be in "." (usually one of your directories). Commands developed at U.C. Berkeley reside in "/usr/ucb", while commands developed at Bell Laboratories live in "/bin" and "/usr/bin".

A number of locally developed programs on the system live in the directory "/usr/local". If you wish all commands that you invoke to have access to these new programs, you can place the command

```
set path=(. /usr/ucb /bin /usr/bin /usr/local)
```

in your ".cshrc" file in your home directory. Try doing this; then **logout** and **login** and do:

```
% set
```

to see that the value assigned to *path* has changed.

One thing you should be aware of is that the C shell examines each directory which you insert into your path and determines which commands are contained there. Therefore, if commands are added to a directory in your *path* after you have started the C shell, they will not necessarily be found by the C shell. This is true for all but the current





directory "." (which the C shell treats specially). If you wish to use a command which has been added in this way, you should give the command

```
% rehash
```

which will cause it to recompute its internal table of command locations so that it will find the newly added command.

Other useful built-in variables are *home* which shows your home directory and *cwd* which contains your current working directory. The *ignoreeof* variable can be set in your ".login" file to tell the C shell not to exit when it receives an end-of-file from a terminal. The C shell does not care about the value of several variables, such as *ignoreeof*, only whether they are **set** or **unset**. Thus, to set this variable you simply do

```
set ignoreeof
```

and to unset it do

```
unset ignoreeof
```

Finally, some other built-in C shell variables of use are *noclobber* and *mail*. The metasyntax

```
> filename
```

which redirects the standard output of a command to "file" will overwrite and destroy the previous contents of the named file if it already exists. You may accidentally overwrite a file which is valuable. If you would prefer that the C shell not overwrite files in this way, you can input:

```
set noclobber
```

in your ".login" file.

Then, trying to do

```
date > now
```

twice will cause the diagnostic "now: File exists". You could type:

```
date >! now
```

if you really wanted to overwrite the contents of "now". The ">!" is a special metasyntax indicating that clobbering the file is ok.†

### 3.3 History

The C shell can maintain a *history list* into which it places previous command lines. It is possible to use a notation to reuse command lines or words from command lines in forming new commands. This mechanism can be used to repeat previous commands or to correct minor typing mistakes in commands.

The string "!$" refers to the last argument to the previous command. The "!" is the **history** mechanism invocation metacharacter, and the "$" stands for the last argument (similarly, "$" in the text editor stands for the end of the line). For example:

```
% sort list.1 > list.2
% pr !$ | lpr
pr list.2 | lpr
%
```

Notice that the C shell echoed the command, as it would have been typed without use of the **history** mechanism, and executed it.

The "!" **history** metacharacter can be used in conjunction with a command name or part of a command name to repeat a previous command line. Thus, using the example above:

† The space between the "!" and the file name "now" is critical here as "!now" would be an invocation of the history mechanism and have a totally different effect.





```
% !p
pr list.2 | lpr
%
```

repeats the last occurrence of a command line beginning with "p" and executes it. It is also possible to add to the repeated command, such as:.

```
% ls -l chap.1
-rw-rw-rw- 1 bill     680 Apr 13 09:45 chap.1
% chmod 600 !$ ; !l
chmod 600 chap.1 ; ls -l chap.1
-rw------- 1 bill     680 Apr 13 09:45 chap.1
%
```

where the file "chap.1" is listed in the long format, the mode for "chap.1" is changed without retyping the file name by using "!$", and is then listed again using "!l" notation.

The "!!" string can also be used to repeat the immediately preceding command. This notation can be added to as well, for example:

```
% ls -l chap.1 chap.2
-rw------- 1 bill     514 Apr 13 12:03 chap.2
-rw------- 1 bill     839 Apr 13 15:54 chap.3
% !! | lpr
ls -l chap.1 chap.2 | lpr
%
```

Another **history** feature is the "*" which denotes the argument list. For example:

```
% ls -l chap.2 chap.3
-rw-rw-rw- 1 bill     514 Apr 13 12:03 chap.2
-rw-rw-rw- 1 bill     839 Apr 13 15:54 chap.3
% chmod 600 !* ; !!
chmod 600 chap.1 chap.2 ; ls -l chap.1 chap.2
-rw------- 1 bill     514 Apr 13 12:03 chap.2
-rw------- 1 bill     839 Apr 13 15:54 chap.3
%
```

If a command line needs to be changed slightly—perhaps a spelling error or different option, there is a substitute feature in **history** that is

similar to the substitute command in the text editor. For example:

```
% ls −l chat.1
No match.
% ^t^p
ls −l chap.1
-rw------- 1 bill     680 Apr 13 09:45 chap.1
%
```

replaces the letter "t" with the correct letter "p". The "^" character is used a the substitute delimiters. Note that only first occurrence will be changed, i.e.:

```
% ls −l chat.1 chat.2
chat.1 not found
chat.2 not found
% ^t^p
ls −l chap.1 chat.2
chat.2 not found
-rw------- 1 bill     680 Apr 13 09:45 chap.1
%
```

If you were to type the **history** command, depending on what is set in you ".login" file, it would print a *history list*. For example:

```
 1  % sort list.1 > list.2
 2  % pr list.2 | lpr
 3  % pr list.2 | lpr
 4  % ls −l chap.1
 5  % chmod 600 chap.1 ; ls −l chap.1
 6  % ls −l chap.1 chap.2
 7  % ls −l chap.1 chap.2 | lpr
 8  % ls −l chap.2 chap.3
 9  % chmod 600 chap.1 chap.2 ; ls −l chap.1 chap.2
10  % ls −l chat.1
11  % ls −l chap.1
12  % ls −l chat.1 chat.2
13  % ls −l chap.1 chat.2
```

Each command line is numbered consecutively starting with 1 (your first command to the C shell) continuing until you **logout**. This number can be used with the "!" character to repeat commands, such as:





```
% !11
ls −l chap.1
-rw------- 1 bill     680 Apr 13 09:45 chap.1
%
```

The **history** mechanism not only saves a lot of typing (especially when you input extremely long command lines), but it is also a very useful reminder of the most recent commands you executed.

Frequently you would just like to look at a previous command without executing it again. The ":p" string can be used to accomplish this, i.e.:

```
% !chmod:p
chmod 600 chap.1 chap.2 ; ls −l chap.1 chap.2
%
```

does not change the mode of the files nor does it list them.

There is a way to refer to a previous command by searching for a string which appeared in it, and there are other less common ways to select arguments to include in a new command. A complete description of all these mechanisms is given in the **csh**(1) entry in the *UniPlus+ System V User's Manual.*

### 3.4 Aliases

The C shell has an **alias** mechanism which can be used to make transformations on input commands. This mechanism can be used to simplify the commands you type, to supply default arguments to commands, or to perform transformations on commands and their arguments. The **alias** facility is similar to a macro facility. Some of the features obtained by aliasing can be obtained also using C shell command files, but these take place in another instance of the C shell and cannot directly affect the current shell's environment or involve commands such as **cd** which must be done in the current shell.

As an example, suppose that there is a new version of the **mail** program on the system called **newmail** you wish to use, rather than the standard program which is called **mail**. If you place the C shell command:

```
alias mail newmail
```

in your ".cshrc" file, the C shell will transform an input line of the form

```
% mail bill
```

into

```
% newmail bill
```

Suppose you prefer the command ls to always show sizes of files, that is to always use the –s option. You can input:

```
alias ls ls –s
```

or perhaps:

```
alias dir ls –s
```

creating a new command syntax **dir** which does an "**ls –s**". If you were to then input:

```
% dir ~bill
```

the C shell will translate this to

```
% ls –s /usr/bill
```

Thus, the **alias** mechanism can be used to provide short names for commands, to provide default arguments, and to define new short commands in terms of other commands. It is also possible to define aliases which contain multiple commands or pipelines that also show where the arguments to the original command are to be substituted. This is accomplished by using the facilities of the **history** mechanism. Thus, the definition:

```
alias cd 'cd \!* ; ls '
```

would do an ls command after each change directory **cd** command. The entire **alias** definition is enclosed in "'" characters to prevent substitutions from occurring and the character ";" from being recognized as a metacharacter. The "!" is escaped with a "\" to prevent it from being interpreted when the **alias** command is typed in. The "\!*" substitutes the entire argument list to the pre-aliasing **cd** command without giving an error if there were no arguments. The ";" separating commands is used here to indicate that one command is to be done and then the next.





**Warning:** The C shell currently reads the ".cshrc" file each time it starts up. If you place a large number of commands there, the C shell will tend to start slowly. You should try to limit the number of aliases you have to a reasonable number—10 or 15 is reasonable, 50 or 60 will cause a noticeable delay in starting up the C shell and make the system seem sluggish when you execute commands from within the text editor and other programs.

### 3.5 More Redirection

There are a few more notations useful to the user which have not been introduced yet.

In addition to the standard output, commands also have a *diagnostic output* which is normally directed to the terminal even when the standard output is redirected to a file or a pipe. It is occasionally desirable to direct the diagnostic output along with the standard output. For instance, if you want to redirect the output of a long running command into a file and wish to have a record of any error diagnostic it produces, you can input:

```
% command >& file
```

The ">&" tells the C shell to route both the diagnostic output and the standard output into "file". Similarly you can give the command:

```
% command |& lpr
```

to route both standard and diagnostic output through the pipe to the line printer daemon **lpr**.*

Finally, it is possible to use the form:

```
% command >> file
```

to place output at the end of an existing file.†

---

* A command form "**command >&! file**" exists and is used when *noclobber* is set and "file" already exists.

† If *noclobber* is set, then an error will result if "file" does exist, otherwise the C shell will create "file" if it doesn't exist. A form "**command >>! file**" prevents it from being an error for "file" to not exist when *noclobber* is set.

### 3.6 Background and Foreground

If the metacharacter "&" is typed at the end of a command line, then that command line is started as a *background* process. This means that the C shell does not wait for it to complete but immediately prompts and is ready for another command. The process runs in the background at the same time that a *foreground* process is running. The processes continue to be read and executed by the C shell one at a time. Thus:

```
% du > usage &
```

would run the **du** program, put the output into the file "usage" and return immediately with a prompt for the next command without waiting for **du** to finish. The **du** program would continue executing in the background until it finished, even though you can type and execute more commands in the meantime. Background processes are unaffected by any signals from the keyboard like the stop, interrupt, or quit signals mentioned earlier.

Processes are recorded in a table inside the C shell until they terminate. In this table, the C shell remembers the command names, arguments and the **process numbers** of all commands in the process as well as the working directory where the process was started. Each process in the table is either running in the foreground with the C shell waiting for it to terminate or running in the background. Only one process can be running in the foreground at one time, but several process can be running in the background at once. When a process is started in the background using "&", the process numbers of all its (top level) commands, is typed by the C shell before prompting you for another command.

### 3.7 Working Directories

The C shell is always in a particular *working directory*. The **chdir** (change directory) command (its short form **cd** may also be used) changes the working directory of the C shell, that is, changes the directory you are located in.

It is useful to make a directory for each project you wish to work on and to place all files related to that project in that directory. The **mkdir** (make directory) command creates a new directory. The **pwd** (print working directory) command reports the absolute pathname of the working directory of the C shell—that is, the directory you are located





in. Thus, in the example below:

```
% pwd
/usr/bill
% mkdir newpaper
% chdir newpaper
% pwd
/usr/bill/newpaper
%
```

the user has created and moved to the directory "newpaper" where he can place a group of related files.

No matter where you have moved to in a directory hierarchy, you can return to your home directory by typing:

```
% cd
```

with no arguments. The name ".." always means the directory above the current one in the hierarchy, thus:

```
% cd ..
```

changes the C shell's working directory to the one directly above the current one. The name ".." can be used in any pathname, thus:

```
% cd ../programs
```

means change to the directory "programs" contained in the directory above the current one. If you have several directories for different projects under, perhaps, your home directory, this shorthand notation permits you to switch easily between them.

### 3.8 Useful Built-in Commands

We now give a few of the useful built-in commands of the C shell describing how they are used.

The **alias** command described above is used to assign new aliases and to show the existing aliases. With no arguments it prints the current aliases. It may also be given only one argument such as

```
alias ls
```

to show the current alias for, e.g., ls.

The **echo** command prints its arguments. It is often used in *shell scripts* or as an interactive command to see what filename expansions will produce.

The **history** command will show the contents of the *history list*. The numbers given with the **history** events can be used to reference previous events which are difficult to reference using the contextual mechanisms introduced above. There is also a C shell variable called *prompt*. By placing a "!" character as its value, the C shell will substitute the number of the current command in the history list. You can use this number to refer to this command in a **history** substitution. Thus you could

```
set prompt='\! % '
```

Note that the "!" character had to be *escaped* even within "'" characters.

The **logout** command can be used to terminate a **login** C shell which has *ignoreeof* set.

The **rehash** command causes the C shell to recompute a table of command locations. This is necessary if you add a command to a directory in the current shell's search path and wish the C shell to find it. Otherwise the hashing algorithm may tell the C shell that the command wasn't in that directory when the hash table was computed.

The **repeat** command can be used to repeat a command several times. Thus to make 5 copies of the file "one" in the file "five", you could type:

```
% repeat 5 cat one >> five
```

The **setenv** command can be used to set variables in the environment. Thus:

```
% setenv TERM adm3a
```

will set the value of the environment variable TERM to "adm3a". A user program **printenv** exists which will print out the environment. It might then show:





```
% printenv
HOME=/usr/bill
SHELL=/bin/csh
PATH=:/usr/ucb:/bin:/usr/bin:/usr/local
TERM=adm3a
USER=bill
%
```

The **source** command can be used to force the current shell to read commands from a file. Thus

```
% source .cshrc
```

can be used after editing a change to the ".cshrc" file which you wish to take effect before the next time you **login**.

The **time** command can be used to cause a command to be timed no matter how much CPU time it takes. Thus:

```
% time cp /etc/rc /usr/bill/rc
0.0u 0.1s 0:01 8%
% time wc /etc/rc /usr/bill/rc
      52  178  1347 /etc/rc
      52  178  1347 /usr/bill/rc
     104  3562  2694 total
0.1u 0.1s 0:00 13%
%
```

indicates that the **cp** command used a negligible amount of user time (u) and about 1/10th of a system time (s); the elapsed time was 1 second (0:01). The word count command **wc** on the other hand used 0.1 seconds of user time and 0.1 seconds of system time in less than a second of elapsed time. The percentage "13%" indicates that over the period when it was active the command **wc** used an average of 13 percent of the available CPU cycles of the machine.

The **unalias** and **unset** commands can be used to remove aliases and variable definitions from the C shell.

## 4. Shell Control Structures and Command Scripts

It is possible to place commands in files and to cause the C shell to be invoked to read and execute commands from these files. These

*executable* files are called *shell scripts*. This chapter outlines some of the information necessary in writing these scripts.

### 4.1 Make

It is important to note what shell scripts are *not* useful for. There is a program called **make** which is very useful for maintaining a group of related files or performing sets of operations on related files. For instance, a large program consisting of one or more files can have its dependencies described in a "makefile" which contains definitions of the commands used to create these different files. Definitions of the means for printing listings, cleaning up the directory in which the files reside, and installing the resultant programs are easily, and most appropriately placed in this "makefile". This format is superior and preferable to maintaining a group of shell procedures to maintain these files.

Similarly when working on a document a "makefile" may be created which defines how different versions of the document are to be created and which options of the text formatter programs, **nroff** or **troff**, are appropriate.

### 4.2 Invocation and the argv Variable

A **csh** command script may be interpreted by saying

```
% csh script ...
```

where "script" is the name of the file containing a group of **csh** commands and "..." is replaced by a sequence of arguments. The C shell places these arguments in the variable *argv* and then begins to read commands from the script. These parameters are then available through the same mechanisms which are used to reference any other shell variables.

If you make the file "script" executable by doing

```
% chmod 755 script
```

or

```
% chmod +x script
```

and place a C shell comment at the beginning of the shell script (i.e., begin the file with a "#" character), a "/bin/csh" will automatically be





invoked to execute "script" when you type

```
% script
```

If the file does not begin with a "#", then the standard shell "/bin/sh" will be used to execute it. This allows you to convert your older shell scripts to use **csh** at your convenience.

### 4.3 Variable Substitution

After each input line is broken into words and history substitutions are done on it, the input line is parsed into distinct commands. Before each command is executed a mechanism know as *variable substitution* is done on these words. Keyed by the character "$", this substitution replaces the names of variables by their values. Thus:

```
echo $argv
```

when placed in a command script, would cause the current value of the variable *argv* to be echoed to the output of the shell script. It is an error for *argv* to be unset at this point.

A number of notations are provided for accessing components and attributes of variables. The notation

```
$?name
```

expands to "1" if name is set, or to "0" if name is not set. It is the fundamental mechanism used for checking whether particular variables have been assigned values. All other forms of reference to undefined variables cause errors.

The notation

```
$#name
```

expands to the number of elements in the variable *name*. Thus:

```
% set argv=(a b c)
% echo $?argv
1
% echo $#argv
3
% unset argv
% echo $?argv
0
% echo $argv
Undefined variable: argv.
%
```

It is also possible to access the components of a variable which has several values. Thus:

```
$argv[1]
```

gives the first component of *argv*, or in the example above "a". Similarly,

```
$argv[$#argv]
```

would give "c", and

```
$argv[1-2]
```

would give "a b". Other notations useful in shell scripts are

```
$n
```

where *n* is an integer as a shorthand for

```
$argv[n]
```

the *n*th parameter.

One minor difference between "$*n*" and "$argv[*n*]" should be noted here. The form "$argv[*n*]" will yield an error if *n* is not in the range "1-$#argv" while "$n" will never yield an out-of-range subscript error. This is for compatibility with the way older shells handled parameters.

Another important point is that it is never an error to give a subrange of the form "n-"; if there are less than *n* components of the given variable, then no words are substituted. A range of the form "m-n" likewise returns an empty vector without giving an error when *m*





exceeds the number of elements of the given variable, provided the subscript *n* is in range.

The notation:

```
$*
```

is a shorthand for

```
$argv
```

The form

```
$$
```

expands to the process number of the current shell. Since this process number is unique in the system, it can be used in generation of unique temporary file names.

### 4.4 Expressions

In order for interesting shell scripts to be constructed, it must be possible to evaluate expressions in the C shell based on the values of variables. In fact, all the arithmetic operations of the language C are available in the C shell with the same precedence that they have in C. In particular, the operations "==" and "!=" compare strings and the operators "&&" and "||" implement the boolean and/or operations. The special operators "=~" and "!~" are similar to "==" and "!=" except that the string on the right side can have pattern matching characters (like *, ? or []), and the test is whether the string on the left matches the pattern on the right.

The C shell also allows file enquiries of the form:

```
-? filename
```

where "?" is replaced by a number of single characters. For instance, the expression primitive:

```
-e filename
```

tells whether the file "filename" exists. Other primitives test for read (-r), write (-w) and execute (-x) access to the file, whether it is a directory (-d), or has non-zero length (-z).

It is possible to test whether a command terminates normally by a primitive of the form "{command}" which returns true, i.e., "1" if the command succeeds exiting normally with exit status 0, or "0" if the command terminates abnormally or with exit status non-zero. If more detailed information about the execution status of a command is required, it can be executed and the variable "$status" examined in the next command. Since "$status" is set by every command, it is very transient. It can be saved if it is inconvenient to use it only in the single immediately following command.

### 4.5 Sample C Shell Script

A sample C shell script which makes use of the expression mechanism of the C shell and some of its control structure follows:

```
% cat copyc
#
# Copyc copies those C programs in the specified list
# to the directory ~/backup if they differ from the files
# already in ~/backup
#
set noglob
foreach i ($argv)
        if ($i !~ *.c) continue  # not a .c file so do nothing
        if (! -r ~/backup/$i:t) then
                echo $i:t not in backup... not cp\'ed
                continue
        endif
        cmp -s $i ~/backup/$i:t # to set $status
        if ($status != 0) then
                echo new backup of $i
                cp $i ~/backup/$i:t
        endif
end
```

This script makes use of the **foreach** command, which causes the C shell to execute the commands between the **foreach** and the matching **end** for each of the values given between "(" and ")" with the named variable, in this case "i" set to successive values in the list. Within this loop we may use the command **break** to stop executing the loop and **continue** to prematurely terminate one iteration and begin the next. After the **foreach** loop, the iteration variable ("i" in this case) has the value at the last iteration.





We set the variable *noglob* here to prevent filename expansion of the members of *argv*. This is a good idea, in general, if the arguments to a shell script are file names which have already been expanded, or if the arguments may contain filename expansion metacharacters. It is also possible to quote each use of a "$" variable expansion, but this is harder and less reliable.

The other control construct used here is a statement of the form:

```
if ( expression ) then
        command
        ...
endif
```

The placement of the keywords here is *not* flexible due to the current implementation of the C shell.‡

The C shell does have another form of the **if** statement of the form:

```
if ( expression ) command
```

which can be written:

```
if ( expression ) \
        command
```

Here we have escaped the newline for the sake of appearance. The command must not involve "|", "&" or ";" and must not be another control command. The second form requires the final "\" to *immediately* precede the end-of-line.

---

‡ The following two formats are not currently acceptable to the C shell:

```
        if ( expression )       # Won't work!
        then
                command
                ...
        endif
    and
        if ( expression ) then command endif      # Won't work
```

The more general **if** statements above also admit a sequence of **else–if** pairs followed by a single **else** and an **endif**, e.g.:

```
if ( expression ) then
        commands
else if (expression ) then
        commands
...
else
        commands
endif
```

Another important mechanism used in shell scripts is the ":" modifier. We can use the modifier ":r" here to extract a root of a file name or ":e" to extract the extension. Thus, if the variable "i" has the value "/mnt/foo.bar", then

```
% echo $i $i:r $i:e
/mnt/foo.bar /mnt/foo bar
%
```

shows how the ":r" modifier strips off the trailing ".bar" and the the ":e" modifier leaves only the "bar". Other modifiers will take off the last component of a pathname leaving the head ":h" or all but the last component of a pathname leaving the tail ":t". These modifiers are described in the **csh**(1) manual entry. It is also possible to use the *command substitution* mechanism described in the next major section to perform modifications on strings to then reenter the C shell's environment. Since each usage of this mechanism involves the creation of a new process, it is much more expensive to use than the ":" modification mechanism.* Finally, we note that the character "#" lexically introduces a C shell comment in shell scripts (but not from the terminal). All subsequent characters on the input line after a "#" are discarded by the C shell. This character can be quoted using "'" or "\" to place it in an argument word.

---

* It is also important to note that the current implementation of the C shell limits the number of ":" modifiers on a "$" substitution to 1. Thus:

```
% echo $i $i:h:t
/a/b/c /a/b:t
%
```

does not do what one would expect.





### 4.6 Other Control Structures

The C shell also has control structures **while** and **switch** similar to those of C. These take the forms:

```
while ( expression )
        commands
end
```

and

```
switch ( word )
case str1:
        commands
        breaksw
...
case strn:
        commands
        breaksw
default:
        commands
        breaksw
endsw
```

C programmers should note that we use **breaksw** to exit from a **switch** while **break** exits a **while** or **foreach** loop. A common mistake to make in csh scripts is to use **break** rather than **breaksw** in switches.

Finally, csh allows a **goto** statement, with labels looking like they do in C, i.e.:

```
loop:
        commands
        goto loop
```

### 4.7 Supplying Input to Commands

Commands run from shell scripts receive (by default) the standard input of the shell which is running the script. This allows shell scripts to fully participate in pipelines, but requires that there be extra notation for commands which are to take inline data.

Thus, we need a metanotation for supplying inline data to commands in shell scripts. As an example, consider this script which runs the editor

to delete leading blanks from the lines in each argument file

```
% cat deblank
# deblank -- remove leading blanks
foreach i ($argv)
ed - $i << 'EOF'
1,$s/CONTROL[ ]*//
w
q
'EOF'
end
%
```

The notation "<< 'EOF'" means that the standard input for the **ed** command is to come from the text in the shell script file up to the next line consisting of exactly "'EOF'". The fact that the "EOF" is enclosed in "'" characters (i.e., quoted) causes the C shell to not perform variable substitution on the intervening lines. In general, if any part of the word following the "<<" which the C shell uses to terminate the text to be given to the command is quoted, then these substitutions will not be performed. In this case, since we used the form "1,$" in our editor script, we needed to insure that this "$" was not variable substituted. We could also have insured this by preceding the "$" here with a "\", i.e.:

```
1,\$s/CONTROL[ ]*//
```

but quoting the "EOF" terminator is a more reliable way of achieving the same thing.

### 4.8 Catching Interrupts

If our shell script creates temporary files, we may wish to catch interruptions of the shell script so that we can clean up these files. We can then do:

```
onintr label
```

where "label" is a label in our program. If an interrupt is received, the C shell will do a "**goto label**" and we can remove the temporary files and then do an **exit** command (which is built in to the C shell) to exit from the shell script. If we wish to exit with a non-zero status, we can do:

```
exit(1)
```

e.g., to exit with status "1".





### 4.9 What Else?

There are other features of the C shell useful to writers of shell procedures. The **verbose** and **echo** options and the related **-v** and **-x** command line options can be used to help trace the actions of the C shell. The **-n** option causes the C shell only to read commands and not to execute them and may sometimes be of use.

One other thing to note is that **csh** will not execute shell scripts which do not begin with the character "#", that is shell scripts that do not begin with a comment. Similarly, the "/bin/sh" on your system may well defer to **csh** to interpret shell scripts which begin with "#". This allows shell scripts for both shells to live in harmony.

There is also another quotation mechanism using "`" which allows only some of the expansion mechanisms we have so far discussed to occur on the quoted string and serves to make this string into a single word as "'" does.

## 5. Other, Less Commonly Used, C Shell Features

### 5.1 Loops at the Terminal; Variables as Vectors

It is occasionally useful to use the **foreach** control structure at the terminal to aid in performing a number of similar commands. For instance, if there were three shells in use on one of your systems – "/bin/sh", "/bin/nsh", and "/bin/csh", to count the number of persons using each shell you could issue the commands

```
% grep -c csh$ /etc/passwd
27
% grep -c nsh$ /etc/passwd
128
% grep -c -v sh$ /etc/passwd
430
%
```

Since these commands are very similar it is possible to do this more easily using **foreach**. For example:

```
% foreach i (`-v sh$`)
? grep -c $i /etc/passwd
? end
27
128
430
%
```

Note here that the shell prompts for input with "? " when reading the body of the loop.

Very useful with loops are variables which contain lists of file names or other words. You can, for example, do:

```
% set a=(`ls`)
% echo $a
csh.n csh.rm
% ls
csh.n
csh.rm
% echo $#a
2
%
```

The set command here gave the variable "a" a list of all the file names in the current directory as value. We can then iterate over these names to perform any chosen function.

The output of a command within "`" characters is converted by the shell to a list of words. You could also place that quoted string within "" characters to take each (non-empty) line as a component of the variable, preventing the lines from being split into words at blanks and tabs. A modifier ":x" exists which can be used later to expand each component of the variable into another variable splitting it into separate words at embedded blanks and tabs.

### 5.2 Braces { ... } in Argument Expansion

Another form of filename expansion, alluded to before involves the characters "{" and "}". These characters specify that the contained strings separated by "," are to be consecutively substituted into the containing characters and the results expanded left to right. Thus:





```
A{str1,str2,... strn}B
```

expands to:

```
Astr1B Astr2B ... AstrnB
```

This expansion occurs before the other filename expansions, and may be nested. The results of each expanded string are sorted separately, left to right order being preserved. The resulting file names are not required to exist if no other expansion mechanisms are used. This means that this mechanism can be used to generate arguments which are not file names, but which have common parts.

A typical use of this would be:

```
% mkdir ~/{hdrs,retrofit,csh}
```

to make subdirectories "hdrs", "retrofit" and "csh" in your home directory. This mechanism is most useful when the common prefix is long, i.e.:

```
% chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}
```

### 5.3 Command Substitution

A command enclosed in "`" characters is replaced, just before file names are expanded, by the output from that command. Thus, it is possible to do:

```
set pwd=`pwd`
```

to save the current directory in the variable "pwd" or to do:

```
% ex `grep -l TRACE *.c`
```

to run the editor ex supplying as arguments those files whose names end in ".c" which have the string "TRACE" in them. †

† Command expansion also occurs in input redirected with "<<" and within "" quotations. Refer to the csh(1) manual entry for full details.

### 5.4 Other Details Not Covered Here

In particular circumstances it may be necessary to know the exact nature and order of different substitutions performed by the C shell. The exact meaning of certain combinations of quotations is also occasionally important.

The shell has a number of command line option flags mostly of use in writing programs and debugging shell scripts. See the **csh**(1) entry in the *UniPlus+ System V User's Manual.*

## 6. Appendix – Special Characters

The following table lists the special characters of the C shell. A number of these characters also have special meaning in expressions. See the **csh**(1) manual entry for a complete list.

Syntactic metacharacters

| | |
|---|---|
| ; | separates commands to be executed sequentially |
| \| | separates commands in a pipeline |
| ( ) | brackets expressions and variable values |
| & | follows commands to be executed in the background |

File name metacharacters

| | |
|---|---|
| / | separates components of a file's pathname; appearing alone, represents the *root* directory |
| . | separates root parts of a file name from extensions |
| ? | expansion character matching any single character |
| * | expansion character matching any sequence of characters |
| [ ] | expansion sequence matching any single character from a set |
| ~ | used at the beginning of a file name to indicate home directories |
| { } | used to specify groups of arguments with common parts |

Quotation metacharacters

| | |
|---|---|
| \ | prevents meta-meaning of following single character |
| ' | prevents meta-meaning of a group of characters |
| " | like ', but allows variable and command expansion |





Input/output metacharacters

| | |
|---|---|
| < | indicates redirected input |
| > | indicates redirected output |

Expansion/substitution metacharacters

| | |
|---|---|
| $ | indicates variable substitution |
| ! | indicates history substitution |
| : | precedes substitution modifiers |
| ^ | used as parameter of history substitution |
| ` | indicates command substitution |

Other metacharacters

| | |
|---|---|
| # | begins scratch file names; indicates C shell comments |
| – | prefixes option (flag) arguments to commands |
| % | prefixes job name specifications |

# Chapter 7
# GLOSSARY

The following alphabetical list defines terms and acronyms used in this guide which may not be familiar to the user.

**.** — The name of your current directory. The character "." is also used in separating components of filenames. At the beginning of a file name, "." is treated specially and not matched by the *filename expansion* metacharacters ?, *, [, and ].

**..** — The name of your parent directory. The command line "**cd ..**" moves you from your present directory to the directory above it.

**alias**— Specifies an abbreviation for a command or series of commands. The C shell has an **alias** command which sets aliases and a command **unalias** to remove aliases.

**argument**— Words following the command on a command line that provide information necessary to execute a program. Command arguments are very often file names.

**ASCII**— American Standard Code for Information Interchange.

**background**— A mode of program execution when the shell does not wait for the command to terminate before prompting for another command. The background can be invoked by ending the command line with an ampersand (&).

**bin**— A directory containing binaries of programs and shell scripts to be executed. The standard *bin* directories are "/bin" containing the most frequently used commands and "/usr/bin," which contains most other user programs.

**builtin command**— A function performed by the shell.

**C language**— A general purpose, low-level programming language used to write programs (such as numerical, text-processing, and data base) and operating systems (such as the UNIX operating system).





**command**— The first word of a command line. It is the name of an executable file that is a compiled program.

**command line**— A sequence of words separated by blanks or tabs typed in by a user. The first words usually specifies the name of a command and the others are arguments to the command.

**command list**— A sequence of one or more simple commands separated or terminated by a new line or a semicolon.

**command procedure**— An executable file that is not a compiled program. It is a call to the shell to read and execute commands contained in a file. A sequence of commands may thus be preserved for repeated use by saving it in a file which can also be called a shell procedure, shell script, a command file, or a runcom according to local preference.

**command substitution**— When the shell reads a command line, any command or commands enclosed between grave accents (`...`) are executed first and the output from these commands replace the whole expression (`...`).

**CONTROL character**— Special character produced by holding down the CONTROL key on your terminal keyboard and simultaneously pressing another character.

**csh**— The command name of the C shell program.

**current working directory**— The current point of reference for accessing data within the file system.

**DELETE**— The DELETE, DEL or RUBOUT key on the terminal keyboard which normally causes an interrupt to be sent to the current job.

**directory**— A structure which contains files that is used to group and organize files and other directories.

**EOF**— The End-Of-File character is the same as an ASCII EOT character.

**EOT**— The End-Of-Text character is generated by holding down the CONTROL key and pressing the lowercase "d" key once. The EOT is used to terminate the shell which usually logs a user off the system.

**erase character**— The character which is used to delete the previous character input on the current line. To turn off the special meaning of the erase character, it must be preceded with a "\." By default, the erase character is #. See **stty**(1) to change the default character.

**escape**— A character "\", when used to prevent the shell from reading the special meaning of a metacharacter, is said to *escape* that character's special meaning. There is also a non-printing keyboard character usually labelled ESC, ESCAPE or ALTMODE that is needed, for example, when using **vi**.

**file**— An organized collection of information containing data, programs, or both which allows users to store, retrieve, and modify information. A simple file name is a sequence of characters other than a slash (/).

**filter**— A command that reads its standard input, transforms it in some way, and prints the result as output.

**flag**— Many commands accept arguments which are not the names of files but are used to modify the action of the commands. These are referred to as *flag* options and consist of one or more letters preceded by the character "−."

**foreground**— A mode of program execution when the shell waits for the command to terminate before prompting for another command.

**full pathname**— The pathname of a specific file starting from the *root* directory (also called *absolute pathname*).

**group identification number (gid)**— A unique number assigned to one or more **login** names that is used to identify groups of related users.





**here documents**— A command procedure that has the form "**command << eofstring**" which causes the shell to read subsequent lines as standard input to the command until a line is read consisting of only the *eofstring*. Any arbitrary string can be used for the *eofstring*.

**history**— The mechanism of the C shell that allows previous commands to be repeated, with or without modification.

**home**— Another name for the **login** directory.

**job**— One or more commands on the same input line separated by "|" or ";" characters.

**keyword parameters**— An argument to a command procedure of the form "**name=value command arg1 arg2 ...**" here *name* is called the keyword parameter. This allows shell variables to be assigned values when a shell procedure is called. The value of *name* in the invoking shell is not affected, but the value is assigned to *name* before execution of the procedure. The arguments (*arg1 arg2 ...*) are available as positional parameters (**$1 $2 ...**).

**kill character**— The character which is used to delete all the characters typed before it on the current line. To turn off the special meaning of the kill character, it must be preceded with a "\." By default, the kill character is @. The default character can be changed via **stty**(1).

**login**— A means by which a user can gain access to the UniPlus$^+$ operating system.

**login name**— A unique string of letters and numbers used to identify a **login**.

**logout**— A procedure to disconnect the user from the UniPlus$^+$ operating system.

**memorandum macros**— The standard general-purpose package of text formatting macros used in conjunction with **nroff** and **troff** to produce documents.

**process**— A program that is in some state of execution. The execution of a computer environment including contents of memory, register values, name of the current directory, status of open files, information recorded at **login** time, and various other items.

**program**— Software that can be executed by a user; a binary file or shell command script which performs a function is called a *program.*

**prompt**— The shell and many programs will print a *prompt* on the terminal when input is expected.

**root**— The directory that is at the top of the entire directory structure.

**script**— Sequences of shell commands placed in a file.

**secondary prompt**— A notification (by default "> ") to the user that the command typed in response to the primary prompt is incomplete.

**sh**— The command name of the shell program.

**shell**— A UniPlus$^+$ system user program written in C language that is is a command language interpreter, i.e., handles the communication between the system and users. The shell accepts commands and causes the appropriate program to be executed.

**shell procedure**— See command procedure.

**standard input**— The standard input of a command is sent to an open file which is normally connected to the keyboard. An argument to the shell of the form "< **file**" opens the specified file as the standard input thus redirecting input to come from the file named instead of the keyboard.

**standard output**— Output produced by most commands is sent to an open file which is normally connected to the printer or screen. This output may be redirected by an argument to the shell of the form "> **file**" which opens the specified file as the standard output.





**metacharacters**— Characters that have a special meaning to the shell, such as < > * ? | & $ ; ( ) \ " ` ' [ ] , etc.

**mode**— An absolute mode is an octal number used in conjunction with **chmod**(1) to change permissions of files.

**nroff**— A text formatting program for driving typewriter-like terminals and printers to produce a screen copy or a hardcopy of a document.

**parent directory**— The directory immediately above another directory. A ".." is the shorthand name for the parent directory. To make the parent directory of your current working directory your new current directory enter the "**cd ..**" command.

**partial pathname**— The pathname between the current working directory and a specific file.

**password**— A string of up to 13 characters chosen from a 64 character alphabet (., \, 0-9, A-Z, a-z).

**pathname**— A sequence of directory names separated by the / character and ending with the name of a file. The pathname defines the connection path between some directory and a file.

**pipe**— A simple way to connect the output of one program to the input of another program, so that each program will run as a sequence of processes.

**pipeline**— A series of filters separated by the character |. The output of each filter becomes the input of the next filter in the line. The last filter in the line will write to its standard output.

**positional parameters**— Arguments supplied with a command procedure that are placed into variable names **$1**, **$2**, ... when the command procedure is invoked by the shell. The name of the file being executed is positional parameter **$0**.

**primary prompt**— A notification (by default "$ ") to the user that the shell is ready to accept another request.

**text editor**— An interactive program (**ed**, **ex** or **vi**) for creating and modifying text, using commands provided by a user at a terminal.

**troff**— A text formatting program for driving a phototypesetter to produce high-quality printed text.

**user-defined variables**— A user variable can be defined using an assignment statement of the form "**name=value**" where *name* must begin with a letter or underscore and may then consist of any sequence of letters, digits, or underscores up to 512 characters. The *name* is the variable. Positional parameters cannot be in the name.

**user identification number (uid)**— A unique number assigned to each **login** that is used to identify users and the owner of information stored on the system.

**variables**— A variable is a name representing a string value. Variables which are normally set only on a command line are called parameters (positional parameters and keyword parameters). Other variables are simply names to which the user (user-defined variables) or the shell itself may assign string values.